Graz University of Technology
Institute for Computer Graphics and Vision

Dissertation

Dynamic Resource Scheduling on Graphics Processors

Markus Steinberger
Graz, Austria, October 2013

Advisor
Prof. Dr. Dieter Schmalstieg
Institute for Computer Graphics and Vision, Graz University of Technology

Referee
Prof. Dr. Jens Krüger
High Performance Computing, University of Duisburg-Essen

To Eva and my parents

The world is parallel.

Joe Armstrong

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, Austria	**September 1, 2013**	
Place	Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, Österreich	**1. September, 2013**	
Ort	Datum	Unterschrift

Abstract

During the last years we have witnessed a severe change in computing. Processors hit the so-called power wall, disallowing them to increase in clock speed. Thus, the arguable only way to feed the ever growing demand for more processing power is parallelism. The highest degree of parallelism currently available in an integrated chip is offered by the graphics processing unit (GPU). A GPU offers tremendous processing power, but does not deliver peak performance, if programs do not offer the ability to be parallelized into thousands of coherently executing threads of execution.

This thesis focuses on solving this issue, unlocking the gates of GPU execution for a new class of algorithms. We present a new processing model enabling fast GPU execution. With our model, dynamic algorithms with various degrees of parallelism at any point during execution are scheduled to be executed efficiently. The core of our processing model is formed by a versatile task scheduler, based on highly efficient queuing strategies. It combines work to be executed by single threads or groups of thread for efficient execution, schedules groups of threads to execute cooperatively and efficiently, and detects harmful execution patterns. It allows different processes to use a single GPU concurrently, dividing the available processing time fairly between them. To assist highly parallel programs, we provide a memory allocator which can serve concurrent requests of tens of thousands of threads. To provide algorithms with the ultimate control over the execution, our execution model supports custom priorities, offering any possible scheduling policy.

With this research, we do not only provide the currently fastest queuing mechanisms for the GPU, the fastest dynamic memory allocator for massively parallel architectures, and the only autonomous GPU scheduling framework that can handle different granularities of parallelism efficiently, we also show the advantages of our model in comparison to state-of-the-art algorithms in the field of rendering, visualization, and geometric modeling. In the field of rendering, we provide algorithms that can significantly speed-up the generation of global illumination computations, assigning more processing power to those image regions that show the lowest image quality. We demonstrate that, with a notion of time, algorithms can be synchronized to high throughput sources like cameras and dynamically adjust their output quality to stay synchronized with the input. In the field of visualization, we analyze three volume rendering techniques and show that detecting and avoiding harmful execution configurations can greatly speed up their execution. In the field of geometric modeling we provide the first GPU-based grammar evaluation system that can generate and render cities in real-time which otherwise take hours to generate and could not fit into the memory.

Kurzfassung

Innerhalb der letzten Jahre begann ein Paradigmenwechsel im Bereich der Computertechnologie. Physikalische Grenzen gestalten eine weitere Erhöhung der Prozessortaktraten zunehmend schwierig. Der einzige Ausweg, um die weiterhin steigende Nachfrage nach immer mehr Rechenleistung bedienen zu können, scheint Parallelisierung zu sein. Ein moderner Grafikprozessor (GPU) stellt derzeit den höchsten Grad an Parallelismus innerhalb eines integrierten Chips zur Verfügung. Die potenziell enorme Rechenleistung einer GPU vermag jedoch nur von Algorithmen genutzt zu werden, die sich in tausende, kohärent arbeitende Ausführungsstränge aufteilen lassen.

Ziel dieser Arbeit ist, die Rechenleistung der GPU einer breiteren Klasse von Algorithmen zur Verfügung zu stellen. Dazu präsentieren wir ein neues Ausführungsmodell, welches die effiziente Abarbeitung von Algorithmen mit inhomogenem, zeitlich veränderlichem Parallelismus ermöglicht. Den Grundstein dieses Modells bildet ein anpassungsfähiger Scheduler, basierend auf hocheffizienten Warteschlangen. Dieser Scheduler kombiniert Arbeitspakete verschiedener Größen zum Zwecke schnellerer, kooperativer Ausführung. Sollte die Ausführung in einen nicht idealen Zustand verfallen, wird dieser detektiert und aufgehoben. Außerdem erlaubt er die gleichzeitige Nutzung der GPU durch mehrere Algorithmen bei fairer Ressourcenaufteilung. Eine detailliertere Kontrolle der Ausführungsreihenfolge innerhalb eines Algorithmus wird durch frei definierbare, dynamische Prioritäten gewährleistet. Um die Unterstützung dynamischer Programme zu komplettieren, stellen wir einen dynamischen Speicherverwalter vor, welcher zehntausende Anfragen gleichzeitig bedient.

Ergebnis unserer Forschung ist nicht nur der zurzeit schnellste Warteschlangenalgorithmus für Grafikprozessoren, der effizienteste Speicherverwalter für massiv parallele Architekturen und der momentan einzige autonome GPU-Scheduler mit Unterstützung verschiedener Granularitäten von Parallelismus, sondern auch eine Weiterentwicklung des momentanen Standes der Technik in den Bereichen Bildsynthese, Visualisierung und geometrischer Algorithmen. So beschleunigt unser Modell die Simulation globaler Beleuchtung durch die Fokussierung der verfügbaren Rechenleistung auf jene Bildbereiche, die den stärksten Beitrag zur Verbesserung der Bildqualität erwarten lassen. Mit unserem Modell ist eine dynamische Anpassung der Ausgabequalität zugunsten der Einhaltung gewisser Echtzeitanforderungen möglich, um sich z.B. mit der hohen Generierungsrate einer Kamera zu synchronisieren. Im Bereich der Visualisierung analysieren wir drei Techniken zur Volumendarstellung und zeigen, wie diese durch intelligentes Scheduling suboptimale Ausführungskonfigurationen vermeiden und signifikant beschleunigt werden. Schlussendlich beschreiben wir die erste Grammatik, die komplette Städte in Echtzeit auf der GPU ableitet und darstellt. Eine Generierung mit traditionellen Methoden würde Stunden benötigen und den verfügbaren Arbeitsspeicher sprengen.

Acknowledgments

I would like to thank everyone who supported me over the last years. First of all, I want to thank my supervisor, Prof. Dieter Schmalstieg, who was always encouraging and provided guidance whenever needed. Dieter is ever full of ideas and never afraid of trying something new. This attitude reflects upon his entire group, which is constantly evolving towards new areas of research. I am very grateful that I got the opportunity to join Dieter's group and collaborate with many talented researchers in a variety of research areas. Special thanks go to my colleagues, who let me participate in their research and contributed to mine. I want to specifically thank Bernhard Kainz, Stefan Hauswiesner, Manuela Waldner, Michael Kenzel, Markus Grabner, Rostislav Khlebnikov, Marc Streit, Alexander Lex, Eduardo Veas, Denis Kalkofen, Raphael Grasset, Stefanie Zollmann, Philip Voglreiter, Bernhard Kerbl, and Gerhard Reitmayr. Furthermore, I want to thank Qiming Hou (Zhejiang University) for providing his implementation of a parallel model file parser, and Christopher Rodrigues (University of Illinois) for providing his GPU memory allocator. Special thanks go to Martin Kenzel for providing me with custom made 3D models when my papers needed some nicer graphics.

This work was supported by the Austrian Science Fund (FWF): P23329.

I am grateful to my parents, Helga and Arnold, who always supported me unconditionally. Since my childhood, they have protected me and have given me the feeling that I could achieve anything I wanted. I would also like to thank my friends for helping me clearing my head from work whenever I needed it. Finally, I want to thank my beloved wife, Eva, for all her love, support, and understanding. I am grateful for every moment I can spend with her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Motivation

Over the last decade, parallel computing has become increasingly available to a wide audience, largely due to the graphics processing unit (GPU), which has evolved into a high-performance, massively-parallel, general-purpose co-processor. GPU hardware is evolving rapidly, steadily increasing the number of parallel processing cores on a single chip, enlarging the available on-device memory, and extending the feature set of previous designs. On the software side, new programming languages such as CUDA have advanced, but the processing model itself has mostly remained untouched since the beginning of general-purpose GPU (GPGPU) programming. This model inherently has several limitations, prohibiting entire classes of algorithms from being executed on the GPU.

This thesis discusses these limitations and shows how resource scheduling strategies designed for massively parallel architectures are able to turn an inflexible GPU into a parallel device capable of executing highly dynamic algorithms. With these advancements, not only can the GPU become capable of executing new classes of algorithms, but also traditional GPU algorithms can execute faster and offer new features.

1.1.1 GPU Programming

To understand the problem of scheduling on the GPU, the differences between CPU and GPU architectures are essential. The key factor for the high performance of the GPU is a massively parallel, throughput oriented design, which is very different from the latency oriented design of the CPU. Nowadays, a single GPU has up to twenty multi-processors, each equipped with a small number of 32-wide single instruction, multiple data (SIMD) units. Altogether, a modern GPU has up to three thousand execution cores. These light-weight cores do not feature expensive logics like branch prediction or out-of-order execution. In combination with the SIMD design, most die space of the GPU can thus be spent on arithmetic units and only little space is needed for control. As data access would quickly become a bottleneck for this architecture, the GPU concurrently executes up to ten times as many threads as there are cores on the device. While one set of threads is waiting for a memory request, another set of threads is chosen for execution. Because there are only limited recourses available on each multi-processor, such as register and fast on-chip shared memory, the number of concurrently executed threads is also limited.

CPU and GPU not only differ in terms of architecture, but also the programming models differ. While programmers are used to write individual sub routines being executed for each thread on a CPU, a function written for a GPU (a so-called *kernel*) is executed by thousands to millions of threads. Although all threads start executing the same function, they are free to choose different execution paths. However, the programmer has to consider the additional constraints of the hardware. As threads are executed in groups (so-called *warps*) on SIMD units, they are only efficient, if all threads within a warp choose the same execution path. A certain number of warps can be grouped to form a so-called block or cooperative thread array (CTA). All warps within a block are executed concurrently on the same multi-processor. They can be synchronized and access fast on-chip shared memory to communicate and work cooperatively. The sum of all blocks forms the kernel.

1.1.2 The First Limitation: Explicit Parallelism

The first limitation of the traditional GPU-programming model is that sufficient data parallelism is required in every stage of an algorithm. These stages are captured by individual kernel function calls. A fixed number of threads is launched for a single kernel and starts executing the same code. There is no way to dynamically adjust the parallelism during a single kernel launch. While from a hardware perspective, it would be sufficient if a high number of coherently executing thread groups are available, good performance using kernel functions can only be achieved, if kernels are started for thousands of threads [56, 103]. This rigid requirement often impairs the straight forward mapping of common algorithms for GPU execution. Such a problem arises, for example, when traversing a tree and executing an operation for every node. One way of parallelizing these classes of algorithms is to subsequently launch a kernel for each tree level. For non-trivial applications, the local tree depth may strongly vary, resulting in underutilization if there are not enough tree nodes available. Additionally, determining the number of nodes per level and mapping them to threads requires synchronization after each level and inefficient parallel reduction methods. The easier and more efficient solution would be to launch new threads for every child node dynamically. Many other algorithms show a similar dependency between control flow and parallelism.

1.1.3 The Second Limitation: Control Switches and Book Keeping

The second limitation is due to the fact that kernel launches in the traditional GPU stream processing model are entirely controlled by the CPU. The idea behind this model is that individual steps can be covered by individual kernels. The output of one kernel serves as input for the next, as outlined in Figure 1.1. As mentioned before, the kernel-based

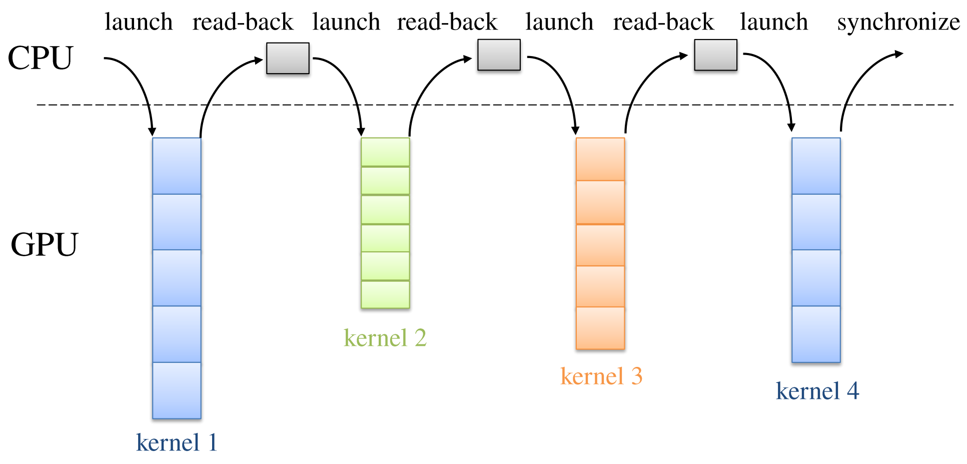


Figure 1.1: In the traditional kernel-based programming model, every algorithm is split up into multiple kernels. Kernel launches are controlled by the CPU, resulting in costly back-and-forth between CPU and GPU.

programming model thus only supports tasks which show a high degree of parallelism, each mapping to an individual kernel launch.

But even for such large tasks, this CPU-controlled programming model can be ineffective. Each kernel launch is associated with multiple sources of overhead: first, GPU occupancy might drop between two successive kernel launches. As a second kernel can only be launched after all blocks of the previous kernel have been finished (to ensure all data is ready), a small number of long-running threads can significantly delay the execution of the following kernel. Second, data which is output by one kernel and consumed by another always has to go through slow global memory. Third, if the number of blocks to be launched depends on the output of the previous kernel, a memory transaction from GPU memory to CPU memory needs to be executed. Such a memory transaction always reduces the GPU utilization to 0%.

While there was no functionality to start the execution of work on the GPU during the initial stage of this thesis, recently this functionality was added to CUDA and named *dynamic parallelism*. With dynamic parallelism, the GPU vendors reacted to the problem that kernels were solely controlled by the CPU. Using dynamic parallelism, it is now possible to start new kernels directly from kernels running on the GPU, as shown in Fig. 1.2. Although, the delay for launching a new kernel from the GPU is about the same as from the CPU, execution can be set up more efficiently using dynamic parallelism. In cases where one would have to read back information for launching the next kernel, the GPU utilization can now be kept up.

One of the downsides of dynamic parallelism is the overhead introduced by track keeping. To ensure the right behavior of an algorithm, newly launched kernels are associated with the launching kernel. Only if all child kernels have been executed, the parent kernel is marked as finished. In this way, it is possible to synchronize on the parent kernel to register the completion of the entire algorithm. The second downside of dynamic parallelism is

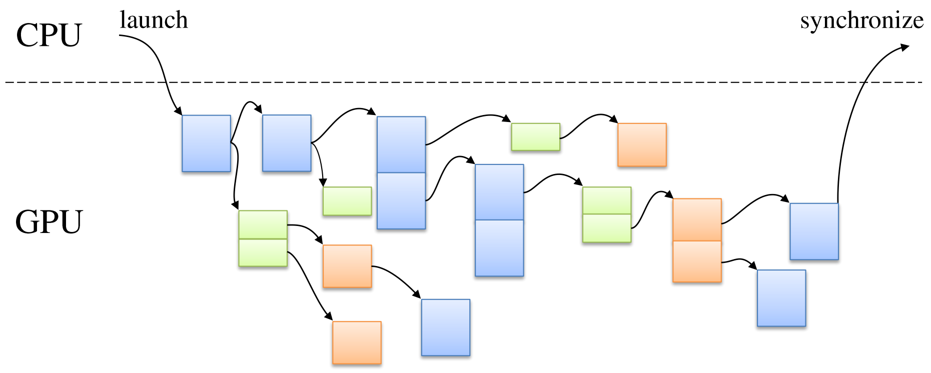


Figure 1.2: With dynamic parallelism kernels can be started on the GPU, keeping track, however, introduces a serious overhead.

that it still only operates on kernels. It is not possible to efficiently initiate the execution of individual threads or warps. A third downside is the fact that communication between different tasks still has to use slow global memory. In the end, algorithms based on dynamic parallelism most often show no benefit over host-controlled kernels. Thus, at this point, this issue cannot be deemed as solved.

1.1.4 The Third Limitation: Influence on Submitted Work

The third limitation is that there is no functionality to influence the execution after starting a kernel. Scheduling on the GPU is currently based on a simple first-in first-out (FIFO) handling of kernels. Thus, work-intensive background tasks can block the entire GPU and delay the execution of high-priority foreground tasks. There is no notion of priorities available. The absence of a priority-based scheduler makes it impossible to use the GPU for tasks with diverse execution characteristics or react on input changes. Additionally, there is currently no way to interrupt or terminate the execution of a running kernel. Interrupting events can only be considered after a kernel has finished. However, if a user changes input parameters – for example in an interactive visualization – the current kernel launch may become obsolete and any further computations rendered useless.

1.1.5 The Fourth Limitation: Dynamic Memory Management

The fourth limitation is the absence of efficient dynamic memory management. Dynamic memory management is an indispensable feature of modern operating systems and virtually every computer program depends on this feature to allow for a dynamic response to varying inputs. The only dynamic memory allocator available on graphics processors comes with the CUDA toolkit. Its internals are unknown, it is unreliable under heavy load and scales badly for high numbers of threads.

1.1.6 The Fifth Limitation: Time Awareness

The fifth limitation is the lack of time-awareness during GPU execution. Currently, neither a common time source between all execution cores on the GPU nor a common time source between GPU and CPU is available. Thus, it is not possible to adjust the algorithm behavior based on the already spent time, the available time until a deadline, or to simply know how much time has passed since a certain event on the CPU occurred. The notion of time is very important for many problems, especially for real-time applications that must fulfill at least soft real-time constraints.

1.2 Objectives

To successfully tackle dynamic resource scheduling on graphics processors, this thesis postulates the following objectives:

O1 Dynamic algorithms are enabled to autonomously execute with high performance on current graphics hardware.

O2 Efficient scheduling of tasks with varying granularities of parallelism becomes possible in a massively parallel environment like graphics processors.

O3 Efficient dynamic memory management becomes possible on graphics processors, even if tens of thousands of threads allocate memory concurrently.

O4 Scheduling on graphics processors becomes controllable and should react on dynamic changes.

O5 An autonomous scheduler on graphics processors can detect execution characteristics which are suboptimal for graphics hardware and regenerate the execution configuration to generate an overall speedup.

O6 A scheduling system fulfilling **O1-O5** will allow a wider range of algorithms to be executed on massively parallel hardware and thus harness the true processing power of the GPU.

One key factor of this thesis is the autonomy of the GPU for executing algorithms, meaning that there is no need for the CPU to guide the execution on the graphics card. However, input data from external devices, user input, and to a certain degree also output has to be coordinated by the CPU. To fulfill **O1**, it must be possible to generate work for an arbitrary number of threads at any point in time. This work must be executed at some point by the requested number of threads. To fulfill **O2**, the overhead introduced by keeping track of the work to be executed, by generating execution patterns that fit the GPU, and by starting the execution, must be kept small. Essentially, the overall execution must be finished faster than a statically-defined, CPU-controlled execution theme. To fulfill **O3**, a dynamic memory allocator on the GPU must meet the requirements of a CPU-based memory allocator and additionally be able to serve memory request in approximately constant time, independently of the number of threads requesting memory concurrently. To fulfill **O4**, the scheduler must be controllable by user level code, *i.e.*, it must support individual dynamically changing priorities, pausing and stopping of individual tasks, and dynamically adjusting an algorithms to the available remaining time. To fulfill **O5**, a scheduler must detect underutilization of individual execution cores, caused by the chosen execution configuration, stop threads causing this deficiency, find a better execution configuration and restart the execution in a better configuration. The gain of the better execution configuration must outweigh the overhead of the management tasks. To fulfill **O6**, there should be at least one algorithm that could previously not be mapped for successful GPU execution, but with the proposed system must be able to execute faster on the GPU than on the CPU.

1.3 Organization of the Thesis

Chapter 1 provides an introduction to the thesis, covering the challenges and goals discussed throughout the work, and points towards the publications which originated from working on this thesis. Chapter 2 discusses the related work in GPU-based task management, queuing strategies usable in multi processor environments, and dynamic memory management.

In the second part of the thesis, the developed algorithms and technologies is presented. Chapter 3 introduces the Softshell processing model, a simple model which supports the definition of tasks with diverse granularities of parallelism and resource requirements. It also provides the design of a three tier dynamic scheduling model, which can be built on top of current GPU designs. While the design specification provides information about a possible system setup, the core components of our implementation are described in the following chapters. Chapter 4 discusses the design of work queues for task management on graphics processors. We compare different blocking and non-blocking queues as well as global and local queues. We evaluate these queues in terms of enqueue and dequeue performance and show which optimizations can strongly enhance all presented queues. Chapter 5 focuses on queue management and priority task management. It investigates how the previously presented queues can be combined to generate different scheduling strategies. Furthermore, we present two strategies to provide priority scheduling mechanism, one for static priorities and the other for dynamically changing priorities. We discuss how different scheduling strategies can be built on top of priority queues and evaluate both priority queues in terms of scheduling accuracy and overhead. Finally, we evaluate these queues in terms of enqueue and dequeue performance as well as execution time for contracting and expanding algorithms with different task characteristics. Chapter 6 discusses different methods of building task schedulers on graphics processors using the previously introduced queues. We show how task schedulers can be built using the traditional host-controlled kernels, a hybrid dynamic parallelism version, and a persistent thread scheduler. Finally, we present a versatile persistent threads scheduler which can execute tasks with different degrees of parallelism. Chapter 7 presents details on memory allocation on the GPU. We discuss the special requirements for memory allocators on the GPU in comparison to CPU allocators and derive a new dynamic memory allocator for the GPU. We compare it to the default CUDA allocator and a doubly linked allocator – the common strategy on the CPU.

The third part of the thesis focuses on applications that can be developed and advanced with the provided scheduling system. Chapter 8 focuses on applications in rendering. The thesis investigates algorithms such as view-dependent mesh simplification, path tracing, image-based visual hull rendering, a GPU X3D parser and a real-time Reyes pipeline. Chapter 9 focuses on applications in volume rendering. It investigates algorithms such as simple front-to-back direct volume rendering, image plane sweep volume illumination, a method for building deep shadow maps on the fly, and particle-based volume rendering, a stochastic volume renderer based on creating and rendering millions of individual particles. Chapter 10 presents the design of a real-time, parallel procedural modeling interpreter. This interpreter is the first to derive context sensitive grammars on a GPU showing speed-ups in comparison to CPU-based derivation.

The final part reflects on the initial objectives and discusses the limitations of the current system and gives directions for future research in the field.

Contribution at a Glance This thesis describes several successful strategies towards different aspects of dynamic resource scheduling on graphics processors. The following list gives an overview over the approaches and achievements of this thesis.

- **The Softshell programming model** is a programming model enabling the execution of tasks with different granularities of parallelism. While standard GPU programming model focuses on massive data parallelism, the Softshell programming model separates the task description, data, and scheduling, and provides an interface for arbitrary degrees of parallelism. We show that this programming model allows to write programs with distributed parallelism in an easier manner than the kernel-based model.

- **The Softshell processing model** presents a three tier model for autonomous task scheduling on the GPU, combining the communication within CPU, GPU, work generation from CPU and GPU, multi-GPU execution, priority scheduling, synchronized timing, and thread re-convergence into a single model.

- **Our distributed-locked queue** is, to the best of our knowledge, the most efficient work queue for massively parallel environments like the GPU. It is up to two magnitudes faster than the fastest queues used by other researchers. We evaluate our queue extensively and show that an efficient queue makes previously used strategies obsolete as the number of thread accessing the queue hardly influences its performance.

- **An extensive analysis and comparison of ten different work queues** shows that blocking queues actually perform better than non-blocking queues if the chance for actually hitting a lock can be reduced and parallel access to the data can be granted to all threads concurrently.

- **Our priority-sorted queue** is the first queuing strategy which provides priority-based scheduling on the GPU. It cannot only handle static priorities but dynamically adjusts the execution order based on a user-defined priority function. Its design reduces addition delays in comparison to simple FIFO queues.

- **Scatter-Alloc** is, to the best of our knowledge, the fastest fully functional memory allocator for the GPU. It serves memory requests in constant time on average, independent of the number of threads concurrently requesting or freeing memory.
- **Our versatile megakernel** is the only megakernel approach which is able to execute different granularities of work, including work for single threads, work for small thread groups, and work for arbitrarily sized blocks. Its design offers high performance for all combinations of granularities while providing the full feature-set known from GPU programming, including the use of shared memory and synchronization.
- **The Softshell framework** is open source and publicly available for everyone to use. It combines all techniques and approaches previously discussed in a simple to use C++ programming interface.
- **Real-time scheduling capabilities** are only possible combining a priority scheduler and a system wide notion of time. We are the first to provide both features. Thus, we are also the first to implement real-time scheduling on the GPU.
- **Our real-time Reyes** implementation using work queues in shared memory is, to the best of our knowledge, the fastest real-time Reyes implementation. It is able to generate and render 100 million micropolygons in every frame with 40 frames per second in Full-HD with four samples per pixel.
- **Our scheduled volume rendering** approaches show that with advanced scheduling, the performance of state-of-the-art volume rendering approaches can be increased significantly, as the utilization of the GPU can be kept high and sub-optimal execution configurations can be avoided.
- **PGA** is the first geometrically context-sensitive grammar that can be executed on the GPU. With our scheduling system, we execute this grammar efficiently. In this way, we are not only the first to bring this kind of grammar evaluation to the GPU, our evaluation system also executes grammars up to 30 to 100 times faster than the state of the art in GPU-based grammar evaluation. We generate geometry even faster than it would take to stream the same amount of geometry to the GPU.

1.4 Individual Publications about this Thesis and Collaboration Statement

Parts of this thesis have been previously published in conferences and journals. The following references describe the preliminary outcome of the work and the contribution of collaborating colleagues.

- **Markus Steinberger**, Michael Kenzel, Bernhard Kerbl, Dieter Schmalstieg:
  Versatile Megakernel
  In review: Transactions on Parallel Computing

  The author of this thesis did most of the implementation and writing of the paper. Michael Kenzel helped with the evaluation and writing of the paper. Bernhard Kerbl implemented the Reyes example. Dieter Schmalstieg revised the paper. This paper was used in Chapter 6.4.

- **Markus Steinberger**, Michael Kenzel, Dieter Schmalstieg:
  Massively Parallel Queuing
  In review: Transactions on Parallel Computing

  The author of this thesis did most of the implementation and writing of the paper. Michael Kenzel helped with the implementation, the evaluation and writing of the paper. Dieter Schmalstieg revised the paper. This paper was used in Chapter 4.

- Rostislav Khlebnikov, Philip Voglreiter, Jaime Garcia, **Markus Steinberger**, Dieter Schmalstieg:
  Parallel Irradiance Caching for Monte-Carlo Volume Rendering
  In review: Eurographics 2014

  Rostislav Khlebnikov, Philip Voglreiter, and Jaime Garcia implemented the Monte-Carlo volume rendering methods, the adaptable octree data structure and the screen space locking mechanism. The author of this thesis worked on the underlying time-based scheduling principals, provided the work queue and the megakernel implementation. The writing of the paper was led by Rostislav Khlebnikov, while the other authors contributed to description of their part of the implementation. Dieter Schmalstieg helped with his expert knowledge and revised the paper.

- **Markus Steinberger**, Michael Kenzel, Bernhard Kainz, Jörg Müller, Peter Wonka, Dieter Schmalstieg:
  Real-time Generation and Rendering of Architecture on the GPU
  In review: Eurographics 2014

  The author of this thesis did the entire implementation of the grammar system, wrote parts of the grammar and wrote most parts of the paper. Michael Kenzel implement the visual effects, helped with the evaluation and writing of the paper. Bernhard Kainz wrote parts of the grammar, helped with the evaluation and revised the paper. Jörg Müller implemented parts of the grammar. Peter Wonka helped with his knowledge and revised the paper. Dieter Schmalstieg provided his experience, wrote parts of the paper and revised the paper. These two papers were used in Chapter 10.

- Philip Voglreiter, **Markus Steinberger**, Rostislav Khlebnikov, Bernhard Kainz, Dieter Schmalstieg:
  Volume Rendering with advanced GPU scheduling strategies,
  IEEE Visualization '13, 2013, poster

Vis '13 Honorable Mention Poster Award

Philip Voglreiter implemented most application cases using Softshell under the supervision of Bernhard Kainz and the author of this thesis. Concurrently with the work of Philip Voglreiter, the author of this thesis improved the Softshell framework to offer the required functionality. The experimentation and writing of the paper were done in cooperative fashion by Philip Voglreiter and the author of the thesis. Dieter Schmalstieg refined the final version of the publication. This work has been used in Chapter 9.

- **Markus Steinberger**, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel and Dieter Schmalstieg:
  Softshell: Dynamic Scheduling on GPUs
  ACM Transactions on Graphics, Proceedings of SIGGRAPH Asia, 2012

  The author of the thesis did most of the implementation work, experimenting and writing of the paper. Bernhard Kainz helped with his experience and was involved in writing of the paper. Bernhard Kerbl implemented the first version of the priority queue and helped with the path tracing example. Stefan Hauswiesner adjusted his visual hull implementation to work with Softshell. Michael Kenzel helped with the mesh simplification example and writing of the paper. Dieter Schmalstieg helped with his experience and revised the paper. Contents of this work has been used in Chapter 1, 3, 8.

- Philip Voglreiter, **Markus Steinberger**, Dieter Schmalstieg, Bernhard Kainz:
  Volumetric Real-Time Particle-Based Representation of Large Unstructured Tetrahedral Polygon Meshes
  in Proceedings of MICCAI MeshMed'12, 2012

  Philip Voglreiter did most of the implementation and writing of the paper. The author of this thesis helped writing the paper. Dieter Schmalstieg helped with his experience and revised the paper. Bernhard Kainz supervised Philip Voglreiter and revised the writing of the paper. This work has been used in Chapter 9.

- **Markus Steinberger**, Michael Kenzel, Bernhard Kainz, Dieter Schmalstieg:
  ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU
  Proceedings of InPar'12, 2012

  The author of this thesis did most of the implementation, writing of the paper and evaluation. Michael Kenzel helped with the base-line allocator, evaluation and writing of the paper. Bernhard Kainz and Dieter Schmalstieg helped by revising the paper. This work has been used in Chapter 7.

- **Markus Steinberger**, Bernhard Kainz, Stefan Hauswiesner, Rostislav Khlebnikov, Denis Kalkofen, Dieter Schmalstieg:

Ray Prioritization Using Stylization and Visual Saliency
Computers & Graphics, 2012

- Bernhard Kainz, **Markus Steinberger**, Stefan Hauswiesner, Rostislav Khlebnikov, and Dieter Schmalstieg:
  Stylization-based ray prioritization for guaranteed frame rates
  NPAR '11 Best Paper Award in Rendering
  in Proceedings of Non-photorealistic Animation and Rendering (NPAR '11), pp. 44-53, 2011

- Bernhard Kainz, **Markus Steinberger**, Stefan Hauswiesner, Rostislav Khlebnikov, Denis Kalkofen, Dieter Schmalstieg:
  Using Perceptual Features to Prioritize Ray-based Image Generation
  in Proceedings of Symposium on Interactive 3D Graphics and Games 2011 (I3D), poster, 2011

  The three above-mentioned publications form a series of publications about the same work. Bernhard Kainz provided the initial idea, large parts of the implementation and most content of the first two publications. The author of this thesis implemented the priority queue, the sampling paper and the saliency-based approach used in the last publication. He also contributed this parts to the paper. Stefan Hauswiesner investigated different image-based GPU reconstruction algorithms and did a large part of the evaluation. Rostislav Khlebnikov revised the paper and helped with the video. Denis Kalkofen and Dieter Schmalstieg added their experience to the content of the paper and the overall system. Parts of this work is used in Chapter 9.

1.5 Additional Publications Beyond the Scope of this Thesis

Besides the work listed in the previous section, the author of this thesis contributed to additional work. This work originates mainly from different projects the author has worked on during the last three years and reflects his interest in work in the fields of visualization, information visualization, user interfaces, and all work using the GPU.

- Rostislav Khlebnikov, Bernhard Kainz, **Markus Steinberger**, Dieter Schmalstieg:
  Noise-based volume rendering for the visualization of multivariate volumetric data
  IEEE TVCG and IEEE Visualization '13, 2013

- Denis Kalkofen, Eduardo Veas, Stefanie Zollmann, **Markus Steinberger**, Dieter Schmalstieg:
  Adaptive Ghosted Views for Augmented Reality
  IEEE ISMAR'13, 2013

- Bernhard Kainz, Stefan Hauswiesner, Gerhard Reitmayr, **Markus Steinberger**, Raphael Grasset, Lukas Gruber, Eduardo Veas, Denis Kalkofen, Hartmut Seichter,

Dieter Schmalstieg:
OmniKinect: Real-Time Dense Volumetric Data Acquisition and Applications
in Symposium On Virtual Reality Software And Technology (VRST), 2012

- Rostislav Khlebnikov, Bernhard Kainz, **Markus Steinberger**, Marc Streit, Dieter Schmalstieg:
Procedural texture synthesis for zoom-independent visualization of multivariate data
in Proceedings of EuroVIS'12, 2012 Dieter Schmalstieg provided guidance and revised the work.

- **Markus Steinberger**, Manuela Waldner, Dieter Schmalstieg:
Interactive Self-Organizing Windows
in Computer Graphics Forum (EG'12), 2012

- Stefan Hauswiesner, Rostislav Khlebnikov, **Markus Steinberger**, Matthias Straka, Gerhard Reitmayr:
Multi-GPU Image-based Visual Hull Rendering
Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization, 2012 Matthias Straka and Gerhard Reitmayr helped with their experience and writing of the paper.

- Manuela Waldner, Raphael Grasset, **Markus Steinberger**, Dieter Schmalstieg:
Display-Adaptive Window Management for Irregular Surfaces
in Proceedings of Interactive Tabletops and Surfaces (ITS'11), 2011

- **Markus Steinberger**, Manuela Waldner, Marc Streit, Alexander Lex, Dieter Schmalstieg:
Context-Preserving Visual Links
InfoVis '11 Best Paper Award
IEEE Transactions on Visualization and Computer Graphics (InfoVis '11), 17(12), 2011

- Manuela Waldner, **Markus Steinberger**, Raphael Grasset, Dieter Schmalstieg:
Importance-Driven Compositing Window Management, *CHI '11 Honorable Mention Award*
in Proceedings of Human Factors in Computing Systems (CHI '11), pp. 959-968, 2011

- **Markus Steinberger**, Markus Grabner:
Wavelet-based Multiresolution Isosurface Rendering
in Proceedings of Eurographics/IEEE VGTC Symposium on Volume Graphics, 2010

Chapter 2

Related Work

Contents

2.1 Scheduling on the GPU

Scheduling for massively parallel, SIMD architectures is a young discipline. Graphics processors have only been available as streaming co-processors for a few years. In 2002, McCool *et al.* introduced *Shader Metaprogramming* [88], which was the first to offer a C++ like programming language in the API to control GPU execution. Buck *et al.* introduced Brook for GPUs [13], which offered the ability to use a GPU as streaming co-processor. Based on this early developments, current GPU programming languages like CUDA [103] or OpenCL [56] were developed. As most early approaches for GPU programming only focused on making the hardware available to programmers, few approaches target scheduling strategies for the GPU, and those that exist are rather limited. However, scheduling is an important and mature problem in operating system design [138]. Unfortunately, GPU architectures differ from CPU architectures in several fundamental properties, making a simple adaption of traditional strategies infeasible.

2.1.1 Built-in GPU scheduling

Built-in GPU scheduling is performed on different levels for GPU architectures as shipped today [100, 102]. The individual blocks of a kernel are distributed to different multiprocessors. On each multiprocessor, a warp scheduler switches between the most suitable warp to be executed [103]. This lowest level of hardware scheduling could possibly be improved to reduce the problem of thread divergence. Dynamic warp formation scheduling as proposed by Fung *et al.* [27, 28] could be used to group threads of different warps to be executed coherently. Dynamic warp subdivision [89] could schedule threads for execution while other threads within a warp are still waiting to complete their memory requests. Large warps and a two-level warp scheduler [97] would also allow to execute subsets of warps as soon as they are ready.

2.1.2 Divergence Avoidance

Thread divergence can not only be addressed in hardware, but also during algorithm design. In raytracing, thread divergence occurs when threads choose different paths during the traversal of the hierarchical acceleration data structure. One possible way of addressing this issue is to use speculative traversal [3], in which all threads within the same warp walk coherently through the data structure. Kalojanov *et al.* [54] propose a two-level nested grids as acceleration structure, as this structure can be optimized for low SIMD divergence. Garanzah and Loop [29] employ a sorting step between successive ray casting steps, sorting rays according to their position and orientation. In this way, coherently executing groups of secondary rays are generated. Hobercook *et al.* [43] use sorting to group CUDA shader calls per shader type to reduce divergence. Novak *et al.* [99] investigate divergence in the context of path tracing. They propose to re-generate new paths for threads that become inactive through Russian Roulette termination. Antwerpen *et al.* [143] and Wald [144] tackle divergence with active thread compaction, *i.e.*, merging partially filled warps to create fewer warps of higher utilization and fill up empty spots with new warps. However, especially for reactive methods like active thread compaction, an increase in performance is often difficult to achieve, as the overhead for this methods is high and the generated execution configuration might show other inefficiencies, like suboptimal memory access patterns. However, we show in multiple examples that an automatic thread compaction method can increase performance.

2.1.3 Task Scheduling

Task scheduling is one of the most essential problems in distributed and parallel computing. Its optimal solution is NP-hard, and heuristics for scheduling thus received a lot of attention since the early times of computing [25, 129, 151]. Task scheduling becomes even harder when moving from a multiprocessor system to a multiprogrammed multiprocessors [4], and further to dynamically changing grids [11, 149].

As the traditional stream processing model provided for the GPU lacks basic task scheduling capabilities, researchers started to use so called persistent thread approaches [3] to implement their own scheduling strategies in software. In this approach, threads operate in an infinite loop setup and draw work from a queue in every iteration of the loop. As soon as there is no more work in the queue, threads finish their execution. One of the features that can be established in a persistent thread approach is load balancing via global queuing. Cederman *et al.* [15] were the first to implement a task scheduler on the GPU. In their work they compared four different queuing strategies and found that lock-free approaches that work well on the CPU also work on the GPU. However, they hypothesize that there is a potential to improve these results with load balancing strategies tuned especially for the GPU. Chen *et al.* [17] extend this scheme to support inserting work into GPU queues from the CPU, while worker blocks on the GPU concurrently draw elements from the queues. If multiple queues are used, a task donation scheme as proposed by Tzeng *et al.* [141] can be used. An alternative has been presented by Chatterjee *et al.* [16], who use a work stealing runtime to distribute work between execution blocks.

Another ability of persistent threads approaches is the ability to exploit producer-consumer locality using fast on-chip shared memory. This ability has been exploited for sparse matrix-vector multiplication [6], sorting algorithms [119], and scan algorithms [12, 150]. As the number of active threads is exactly known in persistent thread approaches, GPU-wide synchronization functions can be implemented. This feature can be built using message passing [134], locks [148], or lock-free approaches [148]. A final feature that can be implemented in persistent thread strategies is divergence avoidance to increase GPU utilization [50]. An overview of persistent thread approaches can be found in [33].

While persistent thread approaches have the ability to implement previously unsupported scheduling strategies, a megakernel approach is needed to dynamically switch between tasks without CPU interaction. The probably most well known megakernel framework is Optix [105], a raytracing system supporting arbitrary shade routines to be combined in a megakernel approach.

The most striking problem of all previously proposed persistent threads and megakernel approach is their inability to adapt to different work granularities and efficiently execute diverse tasks. Persistent threads work either on individual threads or warps, but do not allow custom group sizes, tasks of different thread counts, or the use of shared memory and synchronization. One way of dynamically adjusting GPU execution to different workloads is the recently released dynamic parallelism feature [101]. This feature allows to directly start new kernels from the GPU and could possibly be used to start the execution of new tasks. However, the overhead associated with dynamic parallelism often results in slowdowns in comparison the CPU-controlled execution models. In our work, we propose a versatile megakernel which dynamically adjusts itself to the required number of threads while providing higher performance than traditional megakernels, host-controlled kernels, and dynamic parallelism, making our megakernel the fastest, most versatile execution model for dynamic workloads on the GPU.

2.1.4 Multi-GPU and Heterogeneous Systems

Multi-GPU systems achieve scheduling on a higher level, as shown in RenderAnts [154]. RenderAnts uses BSGP [46], which uses the CPU as a synchronization point between dependent execution steps. Although BSGP improves the process of mapping algorithms to the GPU, it does not improve the scheduling on the GPU itself. When writing programs using BSGP, even more kernel calls are created to adjust the number of threads for different execution steps. In a similar manner, Sponge [45] analyzes and optimizes the static graph of StreamIt programs to generate efficient GPU code for various hardware architectures. Besides RenderAnts, many approaches exist that distribute work over multiple GPU nodes [17, 116]. Furthermore, it is also possible to target heterogeneous setups which combine CPU and GPU execution [139]. As the GPU is programmed differently than the CPU, people are striving towards a common language [45] and scheduling systems [1, 26, 126] for mixed CPU/GPU clusters.

GRAMPS [135] describes an architecture combining GPU-like multiprocessors with a low number of CPU-like latency oriented cores. In this abstract rendering architecture, an algorithm is modeled as a directed graph of worker nodes connected by queues. The GRAMPS architecture is very general, but to date, it has only been tested on a CPU architecture [118]. While the basis of our processing model is similar to GRAMPS, our scheduler enables arbitrary dynamic priorities on a fine grained level. In this way, the scheduling strategy can be adapted to the target application.

2.1.5 Real-time Scheduling

Real-time scheduling is necessary for applications that demand certain tasks to be completed by a given deadline. Although time plays an important role in interactive graphics applications, today's GPU programming is focused on throughput rather than time-awareness. If real-time constraints are considered, they are only tracked from the CPU, by choosing which command to send to the GPU next [55]. These approaches can work well for light-weight tasks, but long running tasks can block the GPU and thus impair the application's real-time behavior. For an extensive overview of real-time scheduling, see the work by Sha *et al.* [124]. Our scheduler is the first to enable real-time scheduling in its original sense on the GPU, making scheduling decisions interdependently from the CPU.

2.2 Queuing

One of the most important features for scheduling on parallel systems is queuing. Basically, queuing algorithms can be divided into *blocking* and *non-blocking* algorithms. Blocking algorithms are based on *mutual exclusion* [24]. Whenever a thread of execution needs access to a shared resource, it acquires a lock on the resource, keeping all other threads from accessing the resource until the lock is released. Thus, access to the resource is strictly serialized. Blocking algorithms hold the potential for *deadlock*. Deadlock occurs when multiple threads of execution each wait for access to a resource one of the other threads currently holds a lock to, resulting in all threads indefinitely waiting for one another and the whole system coming to a halt [18, 137]. Special care has to be taken for these problems to be avoided, or detected and resolved at runtime.

A non-blocking algorithm is constructed in a way that no thread can ever cause system-wide execution to permanently halt, *i.e.*, it guarantees *liveness* of the system. In such an algorithm, deadlock can never occur, yet the non-blocking property in general still allows for *livelock* and *starvation*. Livelock is a situation in which threads are still executing but perpetually interrupting each other's access to a shared resource, resulting in no overall progress being made [137]. Livelock is a special case of starvation, where an individual thread's access to a resource is perpetually interrupted by the activity of other threads, yet, in general, other threads are still making progress [137].

Non-blocking algorithms can be further categorized according to different *liveness criteria* into algorithms that are *obstruction-free*, *lock-free*, or *wait-free*, each of these guarantees being stronger than the one before. An algorithm is obstruction-free [42] if each thread is guaranteed to make progress given no interference by other threads. This property is the weakest form of a non-blocking algorithm, it still allows for livelock and starvation. An algorithm is lock-free [85] if at any point in time it is guaranteed that there is at least one thread in the system making progress. Lock-freedom does not allow for livelock, but still allows for starvation. A wait-free algorithm [41] is the strongest form of a non-blocking algorithm. It guarantees that at any point in time, all threads are making progress; neither livelock nor starvation can occur.

2.2.1 Single Producer, Single Consumer

While queues are among the simplest, most fundamental and thus well researched data structures, the need to support concurrent access can greatly complicate the implementation of even the simplest queues. The available literature on concurrent data structures has a strong focus on lock-free algorithms, which are often held as key to performance in concurrent systems. Yet the lock-free property is often achieved through methods in the spirit of *optimistic concurrency control* [65], *i.e.*, methods built upon the assumption of generally low resource contention. As has already been noted by others before [40, 42], the additional cost of substantially increased algorithmic complexity and redundancy of operations can potentially outweigh the benefits of true lock-freedom.

Many of the well known concurrent queuing strategies such as Lamport's original algorithm [68], *FastForward* [30], or *MCRingBuffer* [72] only support a single producer, single consumer scenario. While very useful in architectures that achieve parallelization through pipelining, dynamic load balancing in multiprocessor systems is, by its very nature, a problem involving multiple producers and multiple consumers. Therefore, our main interest lies in queuing strategies that can support multiple producers and consumers. A recent survey of concurrent queuing strategies and evaluation of their performance on the GPU can be found in [14].

The two basic ways to construct a queue are either as a *link-based* or an *array-based* queue. Link-based queues are set up as singly linked lists, *i.e.*, data structures of independently allocated nodes with each node holding a pointer to the next node in the queue. Array-based queues are built on a block of continuously allocated elements, usually operated as a ring buffer. There are also hybrid constructions, most often in the form of a linked list of arrays, and other variations of these basic queue designs.

2.2.2 Link-based Queues

Valois [142] gives one of the first practical lock-free algorithms for a link-based queue using atomic compare-and-swap (CAS). They use a reference counter on each node to ensure that nodes are never reused as long as any thread still refers to it. Problems with the original algorithm were later discovered and corrected by Michael and Scott [92], though the necessary corrections greatly diminish its practical value.

The link-based Michael-Scott queue [93] is among the most popular lock-free concurrent queue implementations. The baskets queue by Hoffman *et al.* [44] presents a variation on the Michael-Scott queue, recognizing the fact that no particular ordering can be defined for elements that are concurrently inserted into the queue and thus, any order is equally valid. As a back-off algorithm, they collect elements that are being inserted concurrently in what they call a basket, another data structure that can handle concurrent insertion more efficiently but does not adhere to a strict FIFO ordering such as, *e.g.*, a stack. The queue itself is then realized as a linked list of baskets.

2.2.3 Array-based Queues

Valois [142] also gives a lock-free algorithm for an array-based queue. It is, however, not very practical, because it relies on a CAS operation on non-aligned memory addresses, which is generally not supported on any hardware. The ring buffer by Shann *et al.* [125] appears to be the first practical lock-free implementation of an array-based queue, also based on CAS. Tsigas *et al.* [140], building upon the work of Shann *et al.*, try to reduce contention by updating parts of the data structure not with every access but only with every m-th access. The parameter m allows for a trade-off between the amount of additional work that has to be performed to recover the current state of the data structure and the cost of contention. Performance can be increased, if m is chosen such that the overhead expected from contention outweighs the expected cost of the necessary additional work.

While link-based queues are often advertised for their performance, as has been pointed out by, *e.g.*, Shann *et al.* [125], one has to be careful not to overlook the fact that part of their reported theoretical performance might be achieved by ignoring part of the problem. As allocation and deallocation of the memory for the queue elements is an integral part of the algorithm, the design of any array-based queue necessarily has to deal with this problem. However, the discussion of link-based queues often simply assumes the existence of a suitable memory allocator. In cases where the memory for queue nodes can be, *e.g.*, statically allocated, link-based queues can, of course, live up to their full potential. But such applications should be considered the exception rather than the rule and, in general, dynamic memory allocation will be required for the nodes of a linked-queue. Yet concurrent dynamic memory allocation is to be regarded a lofty problem in its own right. Especially if one considers the additional restrictions concerning, *e.g.*, reuse of deallocated memory that some link-based queues impose upon the memory allocation scheme, the overhead of memory allocation and deallocation for the queue nodes can grow to become very significant in practice, particularly on the GPU. Many of the known lock-free algorithms for array-based queues also greatly simplify the problem by only allowing for elements that fit into a machine word and therefore can be written to the queue by an atomic CAS, notable examples being the queues by Shann *et al.* and Tsigas *et al.* mentioned above. In practice this often means that an array-based queue will also require dynamic memory allocation as only pointers to the actual data can be stored in the queue directly. Nevertheless, array-based queues have the desirable property that elements are allocated continuously in memory. Enabling coalesced memory access and having the ability to dequeue multiple elements at once can possibly give array-based queues a profound performance advantage on the GPU.

In this thesis we present a blocking linked queues and blocking array-based queues and show that on massively parallel architectures these queues actually work more efficient than lock-free algorithms. We show that by taking the architectural peculiarities of the GPU into account, previously proposed queuing algorithms can be sped up and our blocking queues can be tuned to serve queuing requests in constant time, independently of the number of threads concurrently accessing the queue.

2.3 Memory Management

Dynamic memory allocation is one of the very basic operations in operating systems. While a lot of work has been done on the CPU side to gain more performance and to adapt the memory management mechanisms to newly available hardware features, only little research has been dedicated to the GPU side.

2.3.1 Dynamic Memory Allocation on the CPU

Dynamic memory allocation has been an important topic in CPU literature since the upcoming of the first programming languages. While earlier programming languages, like for example early versions of Fortran, supported only static memory allocation, modern programming languages support multiple allocation schemes ranging from stack allocation to fine-tuned heap allocators provided by the programming environment itself. Besides optimized standard methods, which can be found in modern high-level programming languages, several researchers proposed improvements to these methods. *Doug Lea's Malloc* [71] and a fast multi-threaded version, ptmalloc [31], are well-known examples of such algorithms. They are currently used in the *GNU C++ library*. A good overview and comparison of such methods is given by Wilson *et al.* [146]. As they discuss, the primary goal of early allocation schemes is to minimize memory overhead.

With the introduction of multi-core CPUs, traditional memory allocation turned out to be a serious bottleneck when multiple cores try to allocate memory in parallel. Nicol showed the drawbacks of state-of-the-art memory allocation when it comes to highly parallel and frequent dynamic memory allocation during simulations [98]. To overcome these problems, *parallel heap* implementations have been introduced by Häggander and Lundberg [36]. With the *Hoard* system by Berger and colleagues [8], a reliable and fast solution for multi-threaded dynamic memory allocation has been found. These systems basically combine one global heap with per-processor (*i.e.*, per-thread) heaps for minimal memory consumption and low synchronization costs. However, heap manipulation in such systems often requires locks and atomic operations because threads cannot be sure on which processor they are executed. Altering the Hoard approach leads to *mostly lock-free malloc* [23], which requires only one heap per CPU. *Multi-Processor Restartable Critical Sections* as introduced with *mostly lock-free malloc*, allowed a speed-up by a factor of ten for selected applications. Later, *scalable lock-free dynamic memory allocation* [91] allowed the complete removal of locks. This allocator uses only atomic operations for dynamic memory allocation and is therefore 'immune to deadlocks regardless of scheduling policies and provides async-signal-safety, tolerance to priority inversion, kill-tolerance, and preemption-tolerance, without requiring any special kernel support or incurring performance over-head' [91]. A recent improvement of lock-free dynamic memory allocation can be found in the *McRT-malloc* approach [49], which is designed for the use with *software transactional memory (STM)*. Compared to the above-mentioned approaches, this allocator also avoids atomic operations in most cases and detects balanced transactional allocations to avoid space blowup. Most recent state-of-the-art memory allocators for multi-core CPUs still use one local heap per thread [121, 123].

All of these methods have been designed for systems with low or medium degrees of parallelism and do not scale well to massively parallel computation as realized on a modern GPU. Many approaches require one heap per processor or thread which can work well for a limited number of cores. However, a GPU provides more processing units by an order of magnitude than any modern CPU, therefore one heap per processor would simply introduce too much overhead. Furthermore, fundamental differences between the hardware architectures prevent the direct application of the above-mentioned methods. These differences are briefly outlined in the next section.

2.3.2 Dynamic Memory allocators on the GPU

GPU-based dynamic memory allocators are very rare. Since a GPU relies on parallelism, operations that require serialization can significantly reduce performance. This problem becomes particularly significant if more than one processor tries to write to the same memory address. In case of such memory collisions, the extensive use of mutually exclusive locks and atomic operations is required. A good overview over the main features of stream architectures and their memory allocation and control schemes is given by Ahn [2].

The degree of parallelism of a modern GPU is rarely seen, even in other comparable stream-processor based architectures. Because the single instruction, multiple data (SIMD) model of the GPU in fact forces many stream processors to run the same code in parallel, locking is a problem, as it is heavily used for CPU based dynamic memory allocation methods. Furthermore, the overhead of inter-core communication through main memory becomes a severe bottleneck as shown by Zhou *et al.* [155] and Lauterbach *et al.* [70].

Furthermore, under certain circumstances, locks might not even work correctly on SIMD architectures such as a modern GPU. Diverging threads in a warp are only marked inactive while the other branch executes. Thus, threads acquiring a lock might get caught up in the inherent busy–wait spin–lock, which can prevent the one thread holding the lock from proceeding [78].

To the best of our knowledge, the only published allocator for the GPU is *XMalloc* [48]. To support a faster reuse of memory blocks, XMalloc stores freed blocks in queues and serves allocation requests from these queues before new memory is used. Furthermore, XMalloc introduces an optimization for SIMD architectures, which groups memory requests that are issued concurrently together within a warp. This reduces the workload on globally shared queues. While XMalloc focuses on fast re-use of memory, our allocator scatters allocation requests with the goal of serving memory requests with little to no conflicts.

Chapter 3

Softshell Processing Model

Contents

3.1 Softshell Entities

The Softshell processing model targets architectures comprised of multiple SIMD units, such as those found on current NVIDIA and ATI graphics cards [22, 100] or on Intel's proposed Larrabee architecture [122]. The conventional stream processing model for GPU programming is based on the assumption that input data is laid out entirely in memory prior to the call of a kernel. In this way, the entire stream may be executed in parallel. In contrast, the Softshell processing model builds on a more complete adaptation of the original stream processing model. We assume that applications are dynamic and show unpredictable data-dependent execution paths. In this way, any portion of input data can become available at any point in time. Thus, the parallelism is not required to only be spatial; it may also be temporal: Multiple tasks can run in parallel, and data can be added to arbitrary input streams at any time. In contrast to GRAMPS [135], Softshell also considers cases in which multiple algorithms with different time characteristics run concurrently. Therefore, we allow for arbitrarily changing priorities to enable well tuned scheduling strategies.

To describe the Softshell processing model as outlined in Figure 3.1, we introduce the notion of *items*, *item sets*, *workpackages*, *procedures* and *events*:

- **work items** describe data to be processed by a single thread,
- **item sets** describe data to be processed by a small group of threads,
- **workpackages** define work for a large groups of threads that require synchronization primitive to cooperatively work on the package,
- **Procedures** describe the functions executed for an *item*, *item set*, or *package*.
- **Events** are triggered by the user and initiate the execution by generating *items*, *item sets* and *packages*.

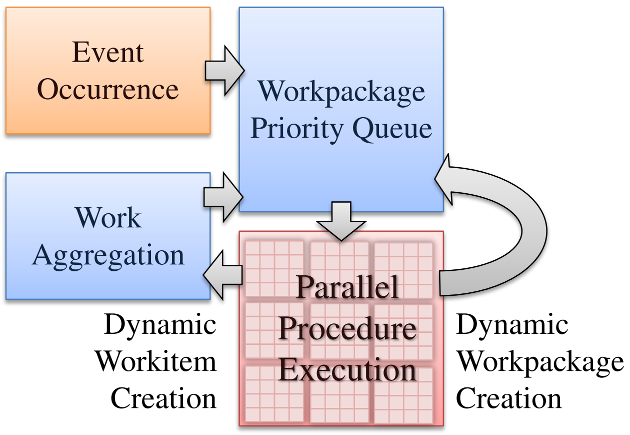

Event Occurrence
Workpackage Priority Queue
Work Aggregation
Parallel Procedure Execution
Dynamic Workitem Creation
Dynamic Workpackage Creation

Figure 3.1: The Softshell processing model: Due to an event, new work becomes available for execution on a GPU. Workpackages are stored in a priority and wait for their parallel execution in a priority queue. Work items and item sets are combined to form groups of sufficient size and join workpackages in the priority queue. When a SIMD processing unit becomes available, it draws the highest priority entry from the queue queue and runs the procedure associated with it. During execution, new work can dynamically be created on the GPU, either in form of entire workpackages or as single work items or item sets to be automatically aggregated. In this way, algorithms with various degrees of parallelism can efficiently be mapped to SIMD architectures.

3.1.1 Procedures

Similar to kernels in the GPU stream processing model, procedures define the execution steps in Softshell. In earlier stream processing literature, a function was called a 'filter', 'agent', or 'sink', depending on its behavior [131]. In contrast to massive kernels, which are intended to occupy the entire GPU with thousands of threads, procedures are designed to be executed by small groups of coherent threads. Thus, a procedure is well suited for the execution on *one* SIMD unit. To fully occupy devices with several SIMD units, Softshell schedules multiple (different) procedures in parallel. As with all function-like constructs, the actual computation steps and delivered results depend on the supplied input parameters. For procedures, these input parameter are formed by items, item sets and workpackages, which are described below. Depending on the input, the active number of threads for a procedure's execution may have to be adjusted by Softshell. To allow for more algorithmic control, the application programmer can demand a fixed number threads to be started for each procedure execution. Softshell expects procedures to run for a short period of time, compared to the execution time of an entire algorithm. Thus, the execution of a procedure will normally not be interrupted until it has been processed and scheduling decisions can be made before starting the next procedure.

3.1.2 Items, Item Sets and Workpackages

Items, item sets and workpackages describe the input data for procedures. While procedures form the description of the execution steps, the combination with items, item sets and workpackages allow Softshell to track what is to be executed. Items, item sets and workpackages capture work of different granularities.

A *work item* is work that should be executed by a single thread. Obviously, no synchronization or communication with other threads is needed or supported. For efficiency, multiple work items of the same kind must be executed together in the same warp to reduce divergence. If different items were mixed, severe slowdowns can be experienced [67].

An *item set* is work that should be executed by x threads, with $2 \leq x \leq$ SIMD-width, *e.g.*, 32 on NVIDIA graphics cards. All threads of an item set must be executed within the same warp to guarantee SIMD lock step execution and allow for working cooperatively. The use of fast on-chip shared memory is allowed and adjusted versions of warp communication functions can also be supported. For efficiency, multiple item sets of the same kind can be grouped and executed on the same SIMD unit if possible. To achieve optimal grouping, the suggested x values are powers of two: 2, 4, 8, 16, and 32. There is no need to offer access to thread synchronization functions, as the SIMD architecture already keeps all threads synchronized.

A *workpackage* is a task that should be executed by an entire block of threads which work on it cooperatively. Similar to a traditional block in a kernel, shared memory and synchronization is supported. We talk about workpackages as soon as their thread count is about the SIMD-width. For efficiency, the number of executing threads should be a multiple of the warp size.

From a programming model point of view, an algorithm can be constructed like a graph with Softshell. Procedures form the nodes of the graph. Work items, item sets and workpackages are added to queues that feed the procedure nodes. Queues belonging to procedures, which require a static number of executing threads, contain fixed sized workpackages. The queues of all other procedures hold workpackages of arbitrary size. These workpackages might be merged while they are waiting for their execution.

Capturing work of different granularities allows Softshell to automatically adjust a trade-off between efficiency and flexibility. During the execution of a procedure, entire workpackages or single items can be generated and queued for arbitrary procedures. If a whole workpackage is generated, Softshell only needs to track a single object to provide work for a whole group of threads. If a procedure only produces a few work items, Softshell may combine them to one workpackage providing a sufficient number of work items for an entire group of threads.

3.1.3 Events

Events initiate the execution of an algorithm in Softshell. This initialization can be explicitly triggered by a user supplying specific input, new data becoming available, or a timer. For these cases the application programmer can create an *event*. When the event is triggered, it queues initial workpackages for a predefined set of procedures. Work created during the execution of these workpackages is again associated with the original event. The event is considered to be processed when all associated workpackages have been processed. In this way, the application can track the progress or cancel the algorithm that is associated with an event.

Former GPU execution models focused on the execution of a single algorithm or tried to achieve the highest throughput for a single pipeline. But real world systems consist of multiple algorithms with severely different execution characteristics. In Softshell, events capture these individual algorithms, which may all fight for the available resources. In comparison to workpackages and procedures, which describe the bits and pieces of an algorithm, an event strings these pieces together and makes the execution of an algorithm traceable.

3.2 Example and Application Programmer Interface

To demonstrate the usage of the basic entities of Softshell, we will describe how an octree-based mesh simplification algorithm can be efficiently mapped to the Softshell processing model. Furthermore, we show some code pieces to ease the understanding of the processing model and to demonstrate that code for Softshell can be written in an intuitive manner (Listing 3.1). One classical mesh simplification approach, which is difficult to optimally map to the GPU, is *hierarchical dynamic simplification (HDS)* [80].

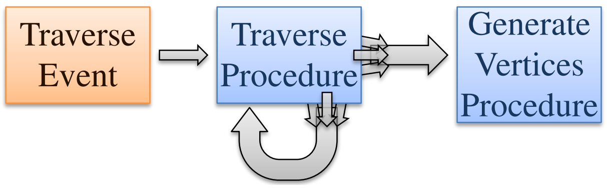


Figure 3.2: Octree-based mesh simplification [80] on the GPU: An event (orange) generates an initial work item. The traverse procedure is executed for every octree node, issuing new work items for the traverse and generate vertices procedure, which are automatically combined by Softshell.

In a preprocessing step, an octree is built and then traversed during rendering. Each node of this octree can either be expanded or collapsed. Expanded nodes increase the local detail by adding triangles to the mesh, while collapsed nodes define vertex positions. The workload for each node may vary strongly, depending on the local complexity of the mesh. This heterogeneous workload can efficiently be handled by the Softshell work aggregation approach, as shown in Figure 3.2. In our implementation, a *traverse event* creates a single work item for the root node to executed by the *traverse procedure.* This procedure creates work items for every child node, which are aggregated and again assigned to the *traverse procedure.* Each time the traversal reaches a boundary node, a work item is issued for the *generate vertex* procedure. Triangles are generated directly in the *traverse procedure.*

The application programming interface (API) essentially exposes the following four classes to the application programmer: a *procedure interface*, a *work interface*, an *event class*, and the *scheduler object*, which is available on the CPU and GPU. To implement a new procedure or to provide a new type of work, one may implement the respective interfaces and register their implementation with the scheduler object. Listing 3.1 shows the simple structures of Softshell for GPU code, while Listing 3.2 shows how the scheduler is controlled from the CPU. More detailed code examples can be found in Appendix A.

The Softshell API provides an efficient control mechanism for multi level parallel execution. With a single command, work for a single thread, an entire group of threads, or multiple groups can be created. Thus, complex algorithms can easily be mapped for GPU execution. Softshell supports a fully customized priority evaluation, whereby arbitrary scheduling strategies can be generated by implementing a single function.

Using the traditional kernel-based model, a large number of intermediate computations are necessary to assign octree nodes to threads. One would normally traverse the octree in a breadth-first fashion, applying a parallel prefix sum to determine the number of active nodes. After that, CPU and GPU must synchronize, so that the correct number of threads is launched for the next level. This strategy is used in current state-of-the art parallel octree traversal solutions [153]. Because this approach only parallelizes the execution of nodes on the same level, the GPU utilization may drop significantly for imbalanced octrees. Using Softshell, nodes from all levels can execute concurrently, dynamically balancing between breadth-first and depth-first traversal. Section 8.1 provides a quantitative comparison of the example outlined here to a traditional kernel-based implementation and shows that the Softshell approach is easier to write and executes more efficiently.

Listing 3.1: The C++ Softshell implementation on top of CUDA is easy to use. For mesh simplification [80], only two procedures are required, as shown in Figure 3.2. Each `Procedure` is created by implementing the `execute` method. Work items for this example correspond to a node identifier only (line 1). Workpackages consisting of multiple work items are defined using templates (line 2). New work items are handed to Softshell using the `issueWorkItem` command (line 19 and 22) and telling the system for which procedure the work item should be queued (template parameter). The `TraverseEvent` issues a single workpackage for the `TraverseProcedure`.

```
 1 typedef uint NodeWorkItem;
 2 typedef CombWorkpackage<NodeWorkItem> NodeWp;
 3
 4 class TraverseProcedure : public Procedure
 5 {
 6 public:
 7  //implement the execute method
 8  __device__ static void execute(Workpackage* workpackage)
 9  {
10   //extract the work item from the package
11   NodeWp* myWp = (NodeWp*)(workpackage);
12   NodeWorkItem myItem = myWp->getMyWorkItem();
13
14   Node* node = getOctreeNode(myItem);
15   for(uint i=0; i < node->numChildren(); ++i)
16   {
17    //check if this node should be expanded
18    if( carryOnTraversal( node->child(i) ) )
19     issueWorkItem<TraverseProcedure>(
20      NodeWorkItem( node->child(i) ) );
21    else
22     issueWorkItem<GenVerticesProcedure>(
23      NodeWorkItem( node->child(i) ) );
24   }
25   writeTriangleOutput(node);
26  }
27 };
28
29 class TEvent
30 {
31  __device__ static void occured()
32  {
33   NodeWorkItem root = 0;
34   issueWorkpackage<TraverseProcedure>
35    ( NodeWp(&root,1) );
36  }
37 };
```

Listing 3.2: Host side API example: A few function calls are sufficient to control the Softshell processing model. Upon initialization, a custom priority evaluation can be specified via template arguments. In this case, the scheduler is configured to execute the `TraverseProcedure` before all others. Subsequently, a custom event is created and triggered once, before the CPU waits for its completion. The associated GPU code is depicted in Listing 3.1.

```
//a custom priority evaluator
struct TraversePriority
{
 __device__ static float
 priority(Procedure* proc, Workpackage* wp)
 {
  //Traverse Procedures should be prioritized
  if(procedureEqual<TraverseProcedure>(*proc))
    return 1.0f;
  else
    return 0.0f;
 }
};

void meshsimplification(DeviceList devices)
{
 //create a scheduler object
 Scheduler scheduler;
 //tell the scheduler to initialize on the chosen set
 //of devices and use the custom priority evaluator
 scheduler.init<TraversePriority>(devices);
 //create a new event of type TEvent
 auto myEvent = scheduler.createEvent<TEvent>();

 //start the event
 myEvent.trigger();
 //wait for the execution on the GPU to finish
 myEvent.join();
}
```

3.3 Three-tier Scheduling

Softshell employs a three-tier scheduling model, which is responsible for distributing work submitted to the system to execution units. The first tier is responsible for work distribution between multiple devices. The second tier is concerned with workpackage priorities and with assigning workpackages to free execution units. The third tier is active during the execution of a single workpackage, addressing diverging threads, pausing, canceling and restarting the active workpackage. The Softshell model only defines the duties of the different scheduling tiers, not their implementation. Nevertheless, we propose an implementation in Section 3.4.

3.3.1 First-tier scheduling

Whenever a new event occurs, the first-tier scheduler is activated. To influence the execution of concurrently active events, events can be assigned priorities. Depending on the state of each GPU, the first-tier scheduler sends an event to the most suitable GPU. For this purpose, it considers the event's priority, the event's execution history, and the current load on each GPU. The first-tier scheduler tries to initiate the event execution in such a way that

(1) high-priority events are completed first,

(2) every GPU is well utilized, and

(3) the execution of all events is completed as soon as possible.

Due to the versatility of the Softshell processing model, the execution time of a single event can only roughly be estimated from previous event occurrences. Thus, the initial choices made by the first-tier scheduler may turn out to be suboptimal, resulting in imbalances in the work loads of different execution devices. As a countermeasure, the first tier scheduler can redistribute workpackages to other execution devices, similar to work stealing, as described in [9]. If necessary, events can be flagged to ensure that all workpackages associated with it are executed on the same device.

3.3.2 Second-tier scheduling

The second tier scheduler is the core of Softshell. In contrast to previous approaches that filled up free processing units with the next best entity, Softshell introduces a more sophisticated logic on this layer: Events and queue elements are considered to have individual priorities. Softshell should schedule the most important elements first, while allowing priorities to change at any point in time. Additionally, every application can define its own per queue element priority function. This priority function is queried by the second-tier scheduler regularly to schedule the highest priority elements first. Obviously, the execution order is independent from the order in which elements are issued.

3.3.3 Third-tier scheduling

The third-tier scheduler is responsible for longer running procedures. It is active during the execution of a single procedure and regularly queries the current execution state. The

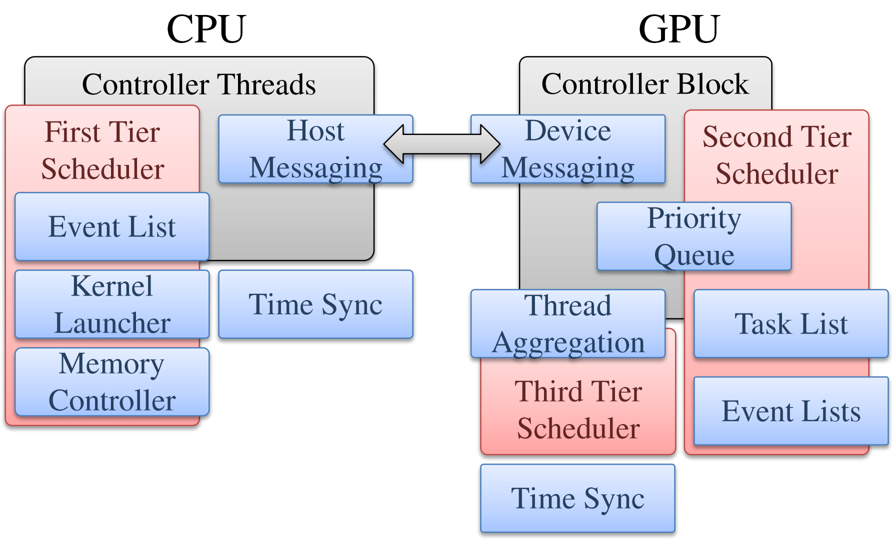


Figure 3.3: Our referecne design of the Softshell processing model is split between CPU and GPU. The CPU part implements the first-tier scheduler and keeps each GPs active. The second- and third-tier schedulers are realized directly on the GPU partially relying on recurring execution within the controller block.

second tier can demand the current execution to stop, if higher priority workpackages became available or the associated event got canceled. If the execution should continue at a later time, the third tier saves all thread contexts and issues the element together with the saved state for later execution. For complex procedures it can be beneficial to regroup diverging threads according to their execution paths. The third-tier scheduler can achieve this by either locally regrouping threads, or by combining threads on a global level. To achieve global regrouping, the third tier stops the executing workpackage and inserts the threads into a global thread aggregation structure, which works similar to work item combination. If threads within this structure cannot be regrouped for a certain time, they are re-issued regardless of their coherency.

3.4 Reference Design

To demonstrate the utility of Softshell, we describe a reference implementation which can be built on top of NVIDIA CUDA. The most important parts of the implementation, which require a more detailed analysis, are described in the Chapters 4, 5, 6, and 7. We replace CUDA's kernel-based interface by a C++ interface. Note that the described design can be made more efficient in the future if implemented as a driver or partially in hardware.

Our design consists of multiple components interacting with each other, as depicted in Figure 3.3. The CPU part of the Softshell design forwards input from the application to the GPU and keeps the internal states synchronized between multiple GPU devices. On the CPU we suggest to assign one thread to each GPU which is responsible for initiating the execution and communicating with the device.

The GPU segment of the implementation is concerned with the execution of work items, item sets and workpackages. On the GPU, we distinguish between *worker blocks* and one *controller block*. The worker blocks are thread blocks, which execute the procedures for all queue elements. The controller block is a thread block, whose only responsibility is to carry out recurring maintenance procedures, including the communication with the CPU and different kinds of scheduling procedures. This setup allows us to react on new events and changing priorities, while the GPU is under full load.

3.4.1 First-tier scheduler

The first-tier scheduler receives events directly from the application. Whenever an event occurs, the scheduler's responsibility is to determine which GPU should execute the event and send an execution request to this GPU with any provided parameters. When determining the most appropriate GPU, the first-tier scheduler regularly uses feedback from each GPU, including the estimated time t_{est} until all events will be processed. The request is forwarded to the GPU with the highest score S:

$$S = \frac{\min\left(n_g, \frac{N}{G}\right)}{\max\left(n_g, \frac{N}{G}\right)} \cdot \frac{\sum_{i=1}^{G} t_{i,est}}{G \cdot t_{g,est}} \cdot \frac{p}{p_{g,max}},$$

where n_g is the number of events active on GPU g; N and G are the overall number of events and graphics cards, respectively; $t_{g,est}$ is the estimated time until all events will be processed on g; p is the priority of the new event; and $p_{g,max}$ is the highest priority among all events currently executing on g. Based on this score, the scheduler assigns a new event to the GPU expected to run out of work first and having the lowest number of events or the GPU that works on low-priority events only. If a GPU is not currently executing any event, the scheduler assigns the new event to that GPU. If multiple graphics cards are out of work, the scheduler favors the GPU with the highest processing power.

We use an **event list** to keep track of all active and previous event executions. Because events can be triggered several times before their first occurrence has entirely been processed, we use a unique identifier per event occurrence. The event list also supports synchronization with the completion of an event and the adjustment of execution parameters. Each event launch can be supplied with a set of parameters, which are made available to the GPU executing the event. As events can occur during the execution, the parameter transfer must happen in parallel with execution. To achieve this goal, we use page-locked host memory, which can be mapped into the address space of the GPU. A **memory controller** takes care of this memory. Whenever a new set of parameters needs to be passed to the GPU, it searches for a free region of sufficient size within the mapped memory. The first fit is then used for the parameter transfer and marked as used until the event execution has finished. This memory can not only be used for a one directional parameter transfer, but also to report results to the CPU. When the event has been fully processed, a callback is executed allowing the application to access data returned from the GPU. After that, the memory is marked as free and can be reused for the next event.

One of the most important features of our design is the **messaging component** that enables a bi-directional communication between the CPU and GPU, while the GPU is active. It is needed to start the execution of events, alter event priorities, report information about event execution, and synchronize the time between the GPU and CPU. Again, we suggest mapped page-locked host memory to establish this communication. Two message queues implemented as ring buffers, one for each direction, serves for this communication. On the CPU side, a mutex can be used to lock the queue when multiple threads want to send messages to the GPU. On the GPU, any running thread can send messages to the CPU, which requires atomic operations to avoid corruption.

As there is no mechanism to automatically react to changes in mapped memory, polling the queue fill levels is the only way to determine changes. For each GPU, one CPU thread should be responsible for polling the message queue, while on the GPU, the controller block checks the queue state. Atomic operations are not supported between GPU and CPU. Therefore, the receiver must not alter the queue state directly when reading messages from a queue. Instead, it can return a message requesting the sender to update the queue state.

3.4.2 Second-tier scheduler

The second-tier scheduler assigns queue elements to worker blocks. Before the second tier becomes active, the initial work descriptors for an event must be created. This action is performed by the controller block, when it receives a message about a new event occurrence. Depending on the algorithm, work elements can either be put directly into queues or allocated with a dynamic memory allocator. Both approaches need in-depth analysis, which we provide in Chapter 4 and 7. To handle the assignment of workpackages to worker blocks, different queuing strategies could be established. The Softshell design specifies an abstract monolithic priority queue that can react on dynamically changing priorities. However, the implementation of an efficient queuing strategy fulfilling these demands requires more in-depth analyses, which we provide in Chapter 4 and 5.

Every entry in this abstract queue holds a reference to the corresponding queue element and procedure. Worker blocks pull elements from the front of the queue, while new elements are inserted at the back. The design of the worker blocks themselves is discussed in Chapter 6. The second tier only becomes active after the execution of a procedure has finished, thus we have to rely on procedures not to occupy worker blocks for too long, to be able to react on newly available high priority elements.

To store meta information of all procedures and call their execute method, we keep a *procedure list*. This meta data includes minimal, maximal, and average execution time and the average number of elements issued by the procedure. This setup enables the estimation of procedure and event execution times. Similarly, we store information about the currently active and previously executed events. Additionally to the average execution time and average number of executed workpackage per event, we also keep track of the number of currently queued elements for this event. If this active element counter drops to zero, the event has been fully processed and the CPU can be informed.

3.4.3 Third-tier scheduler

Due to the lack of GPU hardware support for preemptive multitasking, our current design of the third-tier scheduler uses a cooperative approach. The application programmer

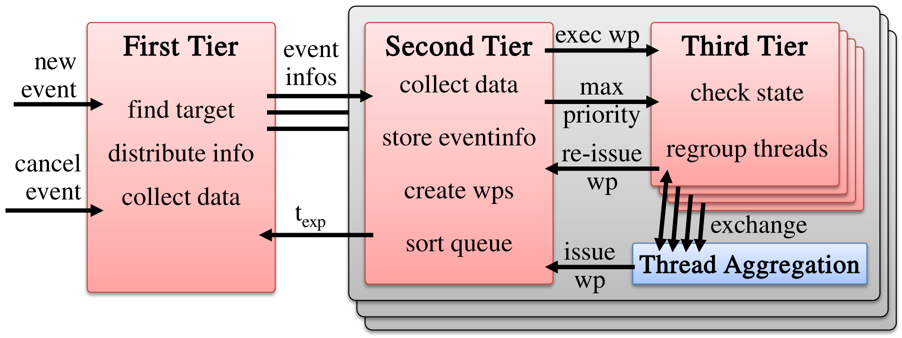


Figure 3.4: Three-tier scheduler: the first tier is activated when an event occurs and forwards it for execution to the most appropriate device. The second tier creates the initial workpackages (wp) for the event and inserts them into the priority queue. The highest-priority elements are forwarded to the third tier for execution by a worker block. The third tier periodically checks the current execution state of all active threads. If the workpackage's priority is too low compared to the front of the queue, it stops execution and re-issues the workpackage. If the threads' execution paths strongly diverge, the third tier regroups threads.

specifies scheduling points, at which the third-tier scheduler is invoked and determines whether execution should continue and whether the thread execution paths are still coherent. Using code analysis for inserting scheduling points is also be possible. However, for our experiments, we decided to let the developer define scheduling points manually and to leave the full control on the user-side. To determine the variables that need to be stored, we use a similar approach to Optix [105] and let the application programmer define the threads payload. In addition to all thread payloads, we also store an identifier for the current execution location. If a block receives a queue element that has previously been executed, it can restore the execution state and continue the execution at the point where it has been suspended. This is similar to kernel relaunching as described by Hou *et al.* [47].

To monitor the thread execution paths, every thread publishes the execution path it will take. If a vote reveals that the thread divergence is above a threshold, the scheduler either stops the execution and submits all threads to the global thread aggregation, or exchanges a few threads with threads already in the thread aggregation. The thread aggregation itself keeps a list of stored threads for all execution points of all procedures. The controller block periodically scans through this list. If there are enough threads stored to form an entirely coherent package or if the threads have been in the list for too long, the controller block merges the threads to form a workpackage and issues it for execution. In the same way, the thread aggregation also concatenates work items to workpackages, as defined in Section 3.1.

3.4.4 Time synchronization

In a system with real-time properties, synchronized time is an important feature for, *e.g.*, estimating the time to completion or enabling scheduling strategies involving a sense of time. We assume that there is at least a counter which keeps pace with the device clock rate on each multiprocessor. This is the current status on NVIDIA graphics cards. To synchronize these counters during system initialization, we use a kernel to write each counter's state to an array in global shared memory. With the messaging component, it is possible to periodically synchronize this array and the time on the CPU using Poincaré-Einstein synchronization [108]. Unfortunately, the device clock rate may not be constant, but might vary with the GPU load. This fact can be compensated by estimating the clock rate between pairs of synchronization messages.

3.4.5 Discussion

The Softshell processing model enables a more sophisticated control of parallel execution than the kernel-based model and provides an intuitive API. Thus, it becomes possible to efficiently map algorithms with a relatively low degree of local parallelism for the execution on massively parallel architectures. Softshell's key features include the ability to generate arbitrary amounts of work directly on the executing device, dynamically adjust priorities for small portions of work, and the ability to control work that has already been submitted to the GPU.

In the following chapters, we present the design and implementation of the core components specified by the Softshell model and discuss our objectives in detail. Chapter 4 discusses the first key feature of Softshell: efficient queuing in massively parallel environments. In this chapter we derive five queues specifically designed for GPU execution. We compare our queues to the best lock-free queues, indicating that on the GPU, our blocking queues are up to 100 times faster than the state-of-the-art in lock-free queuing. In Chapter 5, we discuss how our queues can be used to achieve load balancing between worker blocks, how queues in fast on-chip shared memory can be combined with global queues, and how efficient priority queuing can be accomplished in a massively parallel environment like the GPU. The queuing strategies proposed in this chapter form the basis for Softshell's work aggregation and second tier scheduler, supporting custom priorities. Chapter 6 discussing our versatile megakernel approach, managing the execution within individual worker blocks. Our versatile megakernel is a combination of a persistent threads model, a megakernel, and strategies to dynamically assign warps to the execution of individual procedures. Our versatile megakernel is the first approach that supports varies degrees of parallelism in a single megakernel while still being highly efficient. To complete the description of Softshell's core components, Chapter 7 discusses our highly efficient memory allocator. The memory allocator is not only important to provide algorithms with the ability to react on dynamic input, our previously proposed queues and variants of Softshell's work aggregation heavily rely on memory allocation. In the final chapters of the thesis, we present a variety of use cases demonstrating the usefulness of the Softshell model.

Chapter 4

Massively Parallel Queuing

Contents

4.1 Basic Queues

At the heart of all scheduling strategies are queues which collect and distribute work. Thus, highly-efficient queues are of fundamental importance for minimizing system overhead and maximizing application performance. Current GPU task managers, either use queues based on mutual exclusion, which are already known to perform poorly even on the CPU, or build upon queues which were originally designed for CPU multiprocessor systems. Hence, as mentioned by Cederman *et al.* [15], there is an immediate need to investigate queuing strategies for massively parallel architectures in more detail.

4.2 Queues in Global Memory

A queue serving at the core of a scheduling algorithm on the GPU has to be able to handle thousands of concurrent enqueue and dequeue operations with minimal overhead, while staying withing the bounds of the limited memory allocated to the queue. To aid the following derivation of our queues, we propose five *correctness requirements*, a queue must meet to qualify for the use in such a global scheduling algorithm:

1. **Concurrent insertion and removal:** Any number of threads may concurrently enqueue and dequeue elements. Consistency of the data structure must be ensured at all times.

2. **Read after write safety:** Before an element is dequeued, it must be guaranteed that its entire content has been written by the enqueuing thread.

3. **Write after read safety:** Before the memory of a queue element is reused, it must be guaranteed that all its data has been read.

4. **Overflow protection:** If there is no more space available in the queue, an enqueue operation must fail.

5. **Underflow protection:** If there are no elements in the queue, a dequeue operation must fail.

4.2.1 Collector Queue

Arguably the most simple queue one could possibly build is outlined in Algorithm 1. It consists of a preallocated array, one atomically operated pointer marking the front, and one atomically operated pointer marking the back of the queue. During enqueue, the back pointer is increased using atomic addition, during dequeue, the front pointer is increased using atomic addition. We call this queue the *Collector Queue*, because it can be used to efficiently collect and store intermediate data between kernel launches. By assigning successive queue spots during enqueue and handing out these spots in the same order during dequeue, the Collector Queue partially fulfills requirement (1). However, the other requirements are not met. In its basic form, dequeue operations do not respect the current position of the back pointer and enqueue operations do not respect the current position of the front pointer. Still, with global synchronization between enqueue and dequeue operations as it happens, *e.g.*, between successive kernel launches, the queue can be used. Before a new kernel is launched, the fill-rate of the queue can be determined to make sure an appropriate number of dequeuing threads is started. Although the queue is not safe to be used as work queue in a dynamic load balancing approach, the Collector Queue will serve as a baseline for comparison.

Algorithm 1: Collector Queue

```
$Buffer[QueueSize]$
$Front \leftarrow Back \leftarrow 0$
enqueue (Element)
    $Pos \leftarrow \text{atomicAdd}(Back,1) \mod QueueSize$
    $Buffer[Pos] \leftarrow Element$
    return true
dequeue ( )
    $Pos \leftarrow \text{atomicAdd}(Front,1) \mod QueueSize$
    $Element \leftarrow Buffer[Pos]$
    return Element
```

4.2.2 Mutex Queue

Possibly the simplest queue to fulfill all five requirements is the *Mutex Queue*. Every interaction with the queue is guarded by a global lock, thus, only one thread can modify the queue at a time. We again rely on a fixed size ring buffer with a front and back pointer. To enqueue one or more elements, a thread acquires the lock and computes the number of elements currently in the queue using the front and back pointer. If there is enough space, the element is written to the queue, the back pointer is updated, and the lock is released. To dequeue one or more elements, a thread acquires the lock and, if there are enough elements in the queue, reads the data into shared memory or registers, advances the front pointer, and releases the lock. Similar queues have been used for load balancing by Cederman *et al.* [15] and Tzeng *et al.* [141]. A more efficient version of this queue can be built using separate locks for the front and the back pointer to increase parallelism. Our *Dual Mutex Queue* is outlined in Algorithm 2.

Algorithm 2: Dual Mutex Queue

```
$Buffer[QueueSize]$
$FrontMutex \leftarrow BackMutex \leftarrow Front \leftarrow Back \leftarrow 0$
**enqueue** *(Element)*
    acquire ($BackMutex$)
    **if** size *(Front, Back)* $> QueueSize$ **then**
        $Buffer[Back \mod QueueSize] \leftarrow Element$
        $Back \leftarrow Back + 1$
        release ($BackMutex$)
        **return true**
    release ($BackMutex$)
    **return false**
**dequeue** *( )*
    acquire ($FrontMutex$)
    **if** size *(Front, Back)* $> 0$ **then**
        $Element \leftarrow Buffer[Front \mod QueueSize]$
        $Front \leftarrow Front + 1$
        release ($FrontMutex$)
        **return** $Element$
    release ($FrontMutex$)
    **return false**
```

Assuming correct operation of the lock, all modifications to the queue are strictly linearized, ensuring requirements (1), (2), and (3) hold. Overflow (4) and underflow (5) are avoided by checking the number of elements in the queue before enqueueing or dequeueing elements. While granting at most two of possibly tens of thousands of threads at a time access to the queue ensures correctness, it is, of course, not very efficient.

In addition to not being very efficient, the use of locks is also potentially unsafe on the GPU. The only way to implement a lock on a current GPU, is in the form of a *spin-lock*, *i.e.*, a busy wait periodically trying to acquire the lock. There is no explicit way to put a warp to sleep or even just yield. We identify at least two scenarios in which a busy wait can lead to a livelock situation on the GPU. Assume some threads in a warp try to acquire a lock that is held by another thread of the same warp. If the code is not written very carefully, the threads trying to acquire the lock will end up branching off the thread holding the lock. As execution of different branches has to be serialized, the GPU can end up indefinitely executing only the branch trying to acquire the lock, keeping the thread that holds the lock from releasing it. Similarly, if the thread holding a lock and threads trying to acquire the lock happen to be in-flight on the same multiprocessor, correctness of the algorithm depends on the behavior of the hardware warp scheduler. The only certainty about the hardware warp scheduler is that it always chooses a warp ready for execution. As a warp holding a lock and a warp trying to acquire the same lock, both appear ready to the hardware scheduler, there is no guarantee that the scheduler will ever choose the warp holding the lock for execution. However, exploiting the fact that a global memory access will cause the warp scheduler to switch to another warp, seems to be sufficient to work around the problem of lifelock on all current devices.

4.2.3 CAS Ordered Queue

In order to build a faster queue, multiple threads need to be able to modify the queue simultaneously. Our CAS Ordered Queue uses a ring-buffer as underlying queue storage. Similarly to the Collector Queue, we use an atomically operated front and back pointer to determine the position for the next enqueue and dequeue, leading to concurrent read and write operations. As atomic additions are used to modify the front and back pointer, there is no limit to the number of threads that can concurrently access the CAS Ordered Queue. To circumvent read-before-write and write-before-read hazards, we add a second pair of pointers, called ready-front and ready-back. The front and ready-back pointer mark the range of elements ready for being dequeued, while the back and ready-front pointer mark the range of elements free for being written to, as illustrated in Figure 4.1

Using a second pair of pointers, makes sure that only completely written elements are read (2) and elements are only written to after been read completely (3). It is vital for this strategy that the ready-front and ready-back pointers are updated in a way that ensures connectedness of the ranges of ready and free elements. Because threads do not necessarily complete their operations in the same order they requested queue elements, a simple atomic addition could, *e.g.*, move non-ready elements into the ready range, see Figure 4.1. To avoid this erroneous situation, we delay the increment of the ready pointers until all previous elements in the queue are ready. Instead of a single atomic addition, a thread will continually try to update the respective pointer using an atomic CAS until it succeeds as outlined in Algorithm 3.

To avoid overflow and underflow (4,5), we compute the number of available queue elements from the combination of front and ready-back as well as ready-front and back. This computation can only be carried out after querying a new position by increasing the front pointer for dequeue and increasing the back pointer for enqueue. Thus, special care needs to be taken to leave the pointers in a consistent state when there is no spot available. Using CAS, we only revert the pointers, if they did not change in the meanwhile (or have been reverted by another thread to again match their previous state). Also, a new element could be added while one thread tries to revert the front pointer, or an element could be removed while one thread tries to revert the back pointer. By rechecking the queue state in every revert attempt, we can react to the situation that the fill-rate has changed and make use of a spot that suddenly has become available.

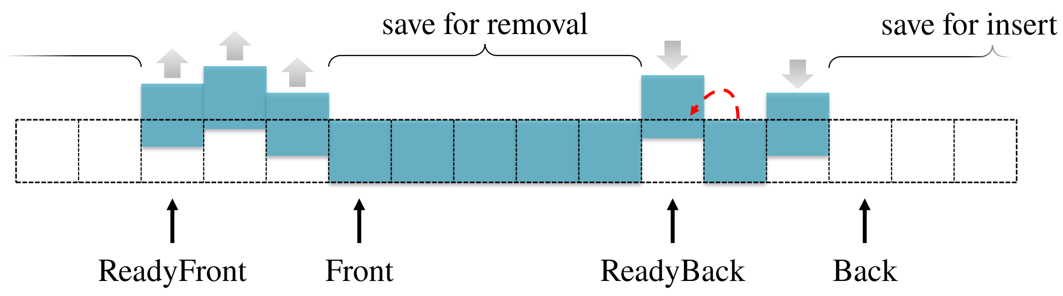


Figure 4.1: The CAS Ordered Queue manages four different sections: current removals, save for removal, current insertions, and save for insert. These sections are manged using four pointers. To update these pointers, the queue relies on atomic CAS operations. If one thread's enqueue or dequeue operation is completed faster than a respective operation on an adjacent element, the CAS operation makes sure that the pointer update is delayed until the slower operations are completed (red).

Algorithm 3: CAS Ordered Queue- red arrow indicates that a thread waiting

```
1  $Buffer[QueueSize]$
2  $Front \leftarrow Back \leftarrow ReadyFront \leftarrow ReadyBack \leftarrow 0$
3  enqueue $(Element)$
4      $Pos \leftarrow$ atomicAdd $(Back, 1)$
5      while $Pos\text{-}ReadyFront \geq QueueSize$ do
6          if atomicCAS $(Back,\ Pos + 1,\ Pos) = Pos + 1$ then
7              return false
8      $Buffer[Pos \mod QueueSize] \leftarrow Element$
9      while atomicCAS $(ReadyBack,\ Pos,\ Pos + 1) \neq Pos$ do
10     return true
11 dequeue $(\,)$
12     $CurrentFront \leftarrow Front$
13     $CurrentReadyBack \leftarrow ReadyBack$
14     while $CurrentReadyBack > CurrentFront$ do
15         if $Pos \leftarrow$ atomicCAS $(Front,\ CurrentFront,\ CurrentFront + 1) =$
           $CurrentFront$ then
16             $Element \leftarrow Buffer[Pos \mod QueueSize]$
17             while atomicCAS $(ReadyFront,\ Pos,\ Pos + 1) \neq Pos$ do
18             return $Element$
19         else
20             $CurrentFront \leftarrow Front$
21             $CurrentReadyBack \leftarrow ReadyBack$
22     return false
```

While the CAS Ordered Queue allows multiple threads to simultaneously write and read elements, it still enforces a strict order on how individual operations can complete. A thread is only allowed to finish its operation if all other threads that started the same operation earlier have already finished. In this way, the queue not only introduces a FIFO ordering among queue elements, but also a FIFO ordering among the threads performing an enqueue or dequeue operation. Keeping this order still introduces some unnecessary delay. The impact of this delay depends on the way warps are scheduled and the exact operation mode of the memory controller. As the queue uses a CAS operation within a loop, it carries a similar risk of lifelock as the Mutex Queue.

4.2.4 Distributed Locks Queue

In order to remove the unnecessary delay introduced by the CAS Ordered Queue, we devise the Distributed Locks Queue, Figure 4.2. This queue again uses a ring buffer and an atomically operated front and back pointer. Individual spots are again assigned to threads using atomic additions on these pointers. To fulfill requirement (2) and (3) we add a flag to each queue element. A zero flag indicates that the spot is unoccupied. When a thread adds an element to the queue, it checks if the flag associated with its element is zero. In case it is not, it spins continuously, waiting for the flag to become zero and the spot to become available. As soon as the spot is marked free, it writes its element to the queue and sets the flag to one (after making sure that the data written is visible to all other threads). If a thread wants to draw an element from the queue, it atomically increases the front pointer and checks if the spot's flag is set to one. If this is not the case, it spins until the flag has been set, and thus the element has become ready for reading. As the queue internally uses a ring buffer, multiple elements can efficiently be dequeued as the front pointer can be increased by an arbitrary number of elements.

Using the atomically operated front and back pointer to determine the fill-level of the queue does not guarantee safe handling of overflow and underflow conditions. As multiple threads may concurrently add elements to the queue, it could happen that the back pointer is increased too far, pointing to positions that are occupied. A thread (A) could determine this problem comparing its position to the front pointer, but it could not safely react on this situation. Waiting until other threads take out elements from the queue could stall the entire device, if all threads happen to run into the same problem. Reducing the back pointer is also unsafe, as a different thread B could increase the back pointer right before A reduces it. At the same time, other threads could dequeue elements from the queue, so that B does not encounter an overflow. In this case, one spot would be left empty (the one before B's spot) and one spot would be used twice (B's spot). Both cases would stall the next thread trying to interact with the spot. To solve these problems, we use an additional atomic counter to hold the queue fill-level, which is increased/decreased before changing the pointers. If the counter is increased too far, the thread can safely decrease the counter, as the pointers have not been changed yet. The principles of the Distributed Locks Queue are outlined in Algorithm 4.

Next to the queue's ability to provide simultaneous access to an arbitrary number of threads, the biggest advantage of the Distributed Locks Queue is that threads are only stalled if the queue is close to being empty or close to being full. In all other cases, there

Algorithm 4: Distributed Locks Queue

```
 1 $Buffer[QueueSize]$
 2 $Flags[QueueSize] \leftarrow \{0, 0, 0, \cdots, 0\}$
 3 $Front \leftarrow Back \leftarrow Count \leftarrow 0$
 4 enqueue $(Element)$
 5     $Num \leftarrow$ atomicAdd $(Count,1)$
 6     if $Num + 1 \geq QueueSize$ then
 7         atomicSub $(Count,1)$
 8         return false
 9     $Pos \leftarrow$ atomicAdd $(Back,1)$ mod $QueueSize$
10     while $Flags[Pos] \neq 0$ do
11     $Buffer[Pos] \leftarrow Element$
12     $Flags[Pos] \leftarrow 1$
13     return true
14 dequeue ( )
15     $Num \leftarrow$ atomicSub $(Count,1)$
16     if $Num < 1$ then
17         atomicAdd $(Count,1)$
18         return false
19     $Pos \leftarrow$ atomicAdd $(Front,1)$ mod $QueueSize$
20     while $Flags[Pos] \neq 1$ do
21     $Element \leftarrow Buffer[Pos]$
22     $Flags[Pos] \leftarrow 0$
23     return $Element$
```

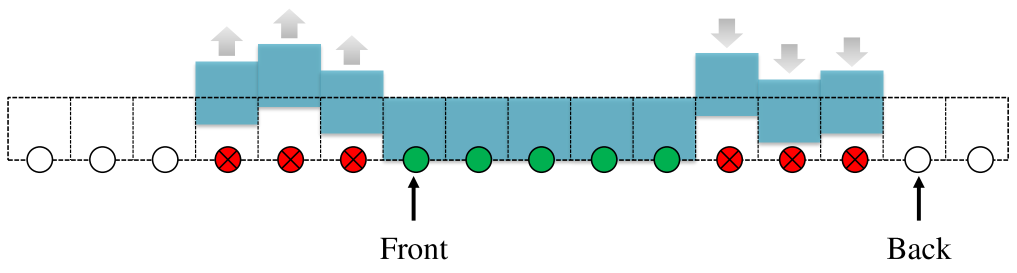


Figure 4.2: The Distributed Locks Queue keeps an individual lock per element. In this way it can make sure that elements are only read after they have been completely written and elements are only written after the spot has been freed.

is sufficient time between adding an element and it being read so that no thread has to wait. An additional advantage is that no more than one thread will ever attempt to change the flag of a specific element. Thus, no atomic operations are required on the locks and memory transactions can be coalesced if multiple threads within a warp enqueue or dequeue elements concurrently. A third advantage of the Distributed Locks Queue is that it can grant threads access to queue elements for a longer time without stalling other threads. This means that instead of copying the data to shared memory or registers, we can simply access the data directly, saving shared memory and registers. When all elements have been processed, the flags of the used elements are erased and the spots can safely be reused by other threads. As long as the queue is not operating at limits of its size, execution of other threads will never be delayed.

One disadvantage of the queue is that it still uses lock-like constructs, with all their associated dangers. However, the number of threads spinning on locks, is limited. The number of threads waiting during a dequeue operation is always smaller or equal to the number currently inserting an element. Analogously, the number of threads waiting during an enqueue operation is always smaller or equal to the number currently removing an element. A second disadvantage is the additional memory required for the locks. As every queue element needs its own lock, additional memory proportional to the size of the queue is needed.

4.2.5 Pointed Queue

Sometimes it might be more reasonable not to store entire elements in the queue, but rather allocate them with a dynamic memory allocator and only store pointers to the actual data in the queue. Such a setup might be advantageous for big elements moving through multiple stages of a pipeline where each stage only changes parts of the element. In this case, we suggest to use a pointed version of the Distributed Locks Queue, as outlined in Algorithm 5. Instead of storing an additional lock for each pointer, one can use the pointer itself as a lock. A zero entry indicates a free spot. As soon as the zero is overwritten, the element is ready for being read. In this way, the memory overhead of the individual locks is avoided and the memory bandwidth for changing the flag can be saved. However, one has to carefully weigh these potential gains against the large overhead of dynamic memory allocation on the GPU.

4.2.6 Linked Queue

Apart from the array-based approach, it is also possible to implement queues in the form of a linked list. While many list-based queues use dummy elements to mark the ends of the list, we simply use a pointer to the first element and a pointer to the last element in the queue, as shown in Figure 4.3. To insert a new element in the queue, the inserting thread allocates a new list node, copies the data to the node, and sets the next pointer of the node to zero. After making sure its writes are visible to all other threads it replaces the current back pointer with a pointer to the new node. If the new element is not the only one in the queue, it sets the next pointer of the node previously at the back of the

Algorithm 5: Pointed Queue

```
$Pointers[QueueSize] \leftarrow \{\textbf{null}, \textbf{null}, \textbf{null}, \cdots, \textbf{null}\}$
$Front \leftarrow Back \leftarrow Count \leftarrow 0$
enqueue ($Element$)
    $Num \leftarrow$ atomicAdd ($Count$,1)
    if $Num + 1 \geq QueueSize$ then
        atomicSub ($Count$,1)
        return false
    $ElementPointer \leftarrow$ alloc ($Element$)
    $Pos \leftarrow$ atomicAdd ($Back$,1) mod $QueueSize$
    while $Pointers[Pos] \neq$ null do
    $Pointers[Pos] \leftarrow ElementPointer$
    return true
dequeue ( )
    $Num \leftarrow$ atomicSub ($Count$,1)
    if $Num < 1$ then
        atomicAdd ($Count$,1)
        return false
    $Pos \leftarrow$ atomicAdd ($Front$,1) mod $QueueSize$
    while $Pointers[Pos] =$ null do
    $ElementPointer \leftarrow Pointers[Pos]$
    $Pointers[Pos] \leftarrow 0$
    $Element \Leftarrow ElementPointer$
    free ($ElementPointer$)
    return $Element$
```

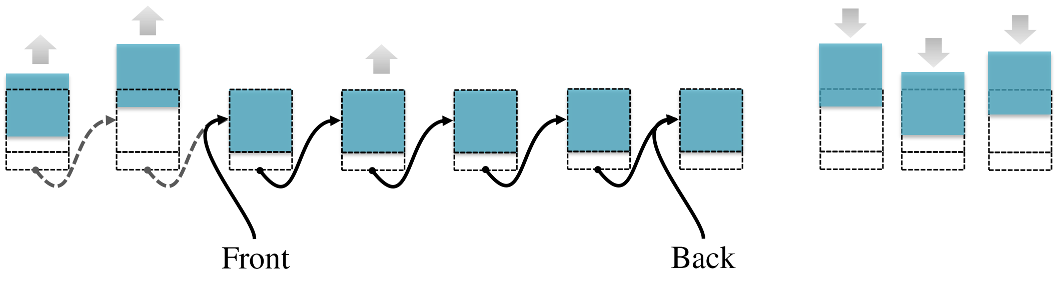


Figure 4.3: The Linked Queue is built as a singly linked list of queue elements.

queue to point to the newly created element. If the back pointer was null, it also sets the front pointer to point to the new node, ensuring consistency of the queue.

If an element should be removed from the queue, the thread repeatedly tries to set the front pointer to the second element in the queue using atomic CAS. If it succeeds, it can safely consume the first element and free the memory allocated for the node. If there is no second element in the queue, the thread tries to remove the last element from the list by resetting the back pointer. In case of success, it also resets the front pointer.

During both, enqueue and dequeue, the back and front pointers may be changed concurrently as the queue is running low on elements. To avoid inconsistencies, we use atomic CAS operations for altering the pointers, always checking for consistency. If an action would interfere with another thread, the thread that detected the problem waits for the other thread to finish its operation. This setup, again, results in a blocking algorithm, but threads will only have to wait if the queue is running low on elements. Note that correctness of this algorithm imposes certain requirements on the memory allocator. The memory of freed nodes must not immediately be reused to ensure that information is not overwritten while other threads are traversing the list. Using a reference counter as proposed by Valois [142] would be one way to implement this behavior.

As insertion involves only the back pointer and removal only the front pointer (as long as there are at least two elements in the queue), requirement (1) can easily be fulfilled. Requirement (2) is fulfilled since elements are only accessed before they are linked in the queue. Reading the data before the node is freed takes care of requirement (3). Requirement (4) is deferred to the memory allocator. As long as it can serve memory requests, new elements can be added to the queue. To limit the queue size, one could additionally use an atomic counter capturing the current fill-rate. Requirement (5) is implicitly met by handling the special case of only one element being left in the queue. If both the front and the back pointer are zero, no element is present.

The major advantage of the Linked Queue is that a new element can be added to the queue with only two global memory operations if the queue is not almost empty. There is no need to retry failed operations or wait for another operation to complete. However, dequeue can be inefficient. In a strategy similar to most non-blocking linked queues, every thread tries to exchange the front pointer with the second element in the queue. Only one thread can be successful and all other threads have to retry. Furthermore, there is no efficient way to read multiple elements at once, removing one element at a time is the only possible way.

4.3 Queues in On-Chip Shared Memory

While the previously discussed queues were all designed to work in global memory, queues for on-chip shared memory have different requirements. First, there is a much tighter memory budget for shared memory than for global memory. Second, the number of threads potentially accessing shared memory concurrently is much lower than in global memory. Third, memory access to shared memory generally will not lead the warp scheduler to switch to another warp. Therefore, spin-locks in shared memory are even more unpredictable than spin-locks in global memory. Fourth, all threads within a block can easily be synchronized, which is exploited by most algorithms on the GPU building around cooperative execution within each block. Efficient block-level synchronization can also be used to introduce and ordering between enqueue and dequeue operations. Still, concurrent enqueue operations of multiple threads need to be supported.

Algorithm 6: Linked Queue

```
$Front \leftarrow Back \leftarrow$ null
enqueue $(Element)$
    $Node \leftarrow$ alloc $(Element)$
    $Node.Next \leftarrow$ null
    $Prev \leftarrow$ atomicExch $(Back, Node)$
    if $Prev \neq$ null then
        $Prev.Next \leftarrow Node$
    else
        while atomicCAS $(Front,$ null$, Node) \neq$ null do
    return true
dequeue ( )
    $F \leftarrow Front$
    if $F =$ null then
        return false
    $N \leftarrow F.next$
    repeat
        if $N \neq$ null then
            $CurrentF \leftarrow$ atomicCAS $(Front, F, N)$
            if $CurrentF = F$ then
                $Element \Leftarrow F.Data$
                free $(F)$
                return $Element$
            if $CurrentF =$ null then
                return false
            else
                $F \leftarrow CurrentF$
                $N \leftarrow CurrentF.next$
        else
            if atomicCAS $(Back, F,$ null$) = F$ then
                while atomicCAS $(Front,F,$ null$) \neq F$ do
                $Element \Leftarrow F.Data$
                free $(F)$
                return $Element$
            else
                $F \leftarrow Front$
                if $F =$ null then
                    return false
                $N \leftarrow F.next$
    until
```

4.3.1 Local Queue

As outlined in Algorithm 7, our standard Local Queue design consists of a fixed size buffer with a single, atomically operated counter. To enqueue an element, we increment the counter to retrieve a free spot. If the queue is out of elements, we reduce the counter again and inform the caller. This simple strategy is sufficient, if synchronization barriers are used to separate enqueue and dequeue. To dequeue elements, we reduce the counter to retrieve a spot and remove elements from the back of the queue. While effectively creating a last in, first out structure, the algorithm is easier to implement, faster, and we find it to be sufficient in most practical applications for small queues in on-chip shared memory.

Algorithm 7: Local Queue

```
$Buffer[QueueSize]$
$Count \leftarrow 0$
enqueue (Element)
    $Pos \leftarrow$ atomicAdd ($Count$,1)
    if $Pos + 1 \geq QueueSize$ then
        atomicSub ($Count$,1)
        return false
    $Buffer[Pos] \leftarrow Element$
    return true
dequeue ( )
    $Pos \leftarrow$ atomicSub ($Count$,1)
    if $Pos$ leq 0 then
        atomicAdd ($Count$,1)
        return false
    $Element \leftarrow Buffer[Pos]$
    return $Element$
```

4.3.2 Dynamic Local Queue

Sometimes it is required to supply individual queues for different procedures. If many different queues need to reside in shared memory, one quickly hits the limits of available shared memory. In these situations it would be beneficial if shared memory could be dynamically assigned to queues as needed. Our Dynamic Local Queue illustrated in Figure 4.4 and Algorithm 8 can provide this feature.

During enqueue, if there is no sub-queue available or all suitable queues are full, we try to allocate a new sub-queue. At first, we decide on how much space the new sub-queue should occupy. Queue size is either limited by the remaining available shared memory or a custom threshold. According to our experiments, a reasonable choice for sub-queues holding work items is to supply space for at least the number work items that can be executed concurrently by a worker block. For workpackages, a good trade-off depends on the size of a workpackage and the chance of a new element being added to the queue.

Algorithm 8: Dynamic Local Queue

```
1  $Buffer[MemSize] \leftarrow \{0, 0, 0, \cdots, 0\}$
2  enqueue (ProcedureId,Element)
3      $HeaderPos \leftarrow 0$
4      while $HeaderPos < MemSize$ do
5          $CurrentSize \leftarrow 0$
6          $CurrentHeader \leftarrow Buffer[HeaderPos]$
7          if lower bits of $CurrentHeader = ProcedureId$ then
8              $CurrentSize \leftarrow$ upper bits of $CurrentHeader$
9          else if $CurrentHeader = 0$ then
10             $CurrentSize \leftarrow$ computeSubQueueSize ($ProcedureId$)
11             $NewHeader \leftarrow (CurrentSize)$ mixed with $ProcedureId$
12             if $CurrentHeader \leftarrow$ atomicCAS (Buffer[HeaderPos],0,NewHeader)
               $\neq 0$ then
13                 if lower bits of $CurrentSizeIdMix = ProcedureId$ then
14                     $CurrentSize \leftarrow$ upper bits of $CurrentSizeIdMix$
15             else
16                 $CurrentHeader \leftarrow NewHeader$
17         if $CurrentSize > 0$ then
18             $Pos \leftarrow$ atomicAdd ($Buffer[HeaderPos$+1],1)
19             if $Pos < CurrentSize$ then
20                 $Buffer[HeaderPos +2+ Pos] \leftarrow Element$
21                 return true
22             else
23                 atomicSub ($Buffer[HeaderPos$+1],1)
24         $HeaderPos \leftarrow$ lower bits of $CurrentHeader$ +2
25     return false
26 dequeue ( )
27     $HeaderPos \leftarrow 0$
28     while $HeaderPos < MemSize$ do
29         if $Buffer[HeaderPos+1] > 0$ then
30             $Pos \leftarrow$ atomicSub ($Buffer[HeaderPos$+1],1)
31             if $Pos$ leq 0 then
32                 atomicAdd ($Buffer[HeaderPos$+1],1)
33             else
34                 $Element \leftarrow Buffer[HeaderPos +2+ Pos]$
35                 return $Element$
36         $HeaderPos \leftarrow HeaderPos +$ lower bits of $Buffer[HeaderPos]$ +2
37     return false
```

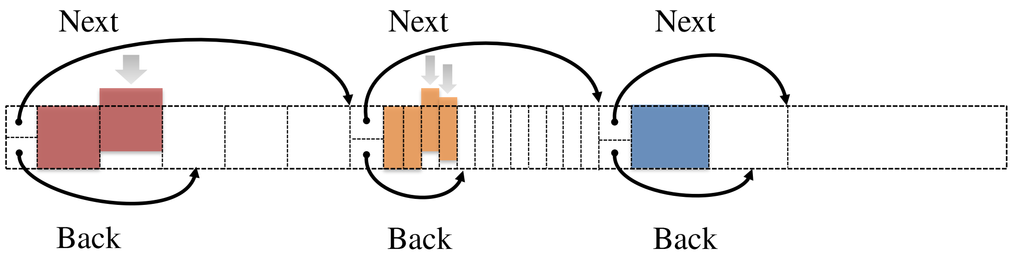


Figure 4.4: The Dynamic Local Queue is able to manage multiple sub-queues in on-chip shared memory dynamically. Every sub-queue keeps a pointer to the next queue and a pointer to the back of the queue.

Right before each sub-queue in shared memory, we store additional header information. The header includes the type of elements the queue holds, the size of the memory block allocated to the queue, a fill-rate counter and possibly some additional information relevant for a particular application. The main challenge of the dynamic queue is that multiple threads can concurrently try to allocate a new sub-queue. As the warp scheduler does not necessarily switch to another warp when accessing shared memory, waiting for another thread to write header information would not be a good strategy. Instead, we initialize the entire dynamically managed shared memory to zero and establish the following, lock-free allocation strategy: Whenever a thread needs a new sub-queue, it generates a 32-bit descriptor holding the sub-queue's required size and an identifier for the type of data the queue will be used for. It then tries to start a new queue by writing this part of the header to the next free location in shared memory using atomic CAS. If it succeeds, the queue is created. If another thread successfully wrote a descriptor in the meanwhile, we check if the thusly allocated queue can serve our request and, in this case, use it. If we cannot use this queue, we extract the queue's size from the header, skip over the new queue, and try again.

If sub-queues are already present in shared memory on an enqueue operation, we check their respective headers for a suitable queue that can still hold elements. If we can find a queue, we simply increment its fill-rate counter and try to insert the element into the queue. If there is not enough space, we decrement the fill-rate counter again and skip to the next queue. If elements are to be drawn from the queues, we check one queue after the other and, when we find a sufficient number of elements in a sub-queue, we draw them from the back of the sub-queue similarly to our non-dynamic Local Queue.

In order to be able to move from one sub-queue to next and avoid fragmentation, we require all sub-queues to be tightly packed. When all elements have been drawn from a sub-queue, we perform a compaction step to remove empty sub-queues. During compaction, we copy all non-empty sub-queues following an empty sub-queue to the front, overwriting the empty queue. The memory freed at the back is cleared with zeros to allow new sub-queues to be allocated there.

While the Dynamic Local Queue provides great versatility, the search for a fitting sub-queue, dynamic allocation of sub-queues, and the necessary compaction step all introduce additional overhead. Therefore, it only makes sense to use this type of queue in scenarios with a large number of different procedures.

4.4 Optimizations

The feature set of modern GPU architectures allows for many small, but very effective optimizations. In the following, we describe a number of optimizations targeted towards devices based on NVIDIA CUDA, which offer access to a large set of low level operations.

4.4.1 Warp Optimization

If multiple threads within a warp intend to perform the same operation on a global data structure, it might be possible to locally combine the effects of the individual operations and then have only one thread modify the global data structure. Such a local combination of the operations of multiple individual threads within the same warp is widely known as *warp optimization* and can greatly help to relieve contention. The thread carrying out the combined operation is usually referred to as the warp's *lead thread*. As threads within the same warp execute in lock-step, warp voting functions can be used to identify all threads that concurrently want to perform an operation. Thus, warp optimization can often be implemented transparently to the user, beneath the interface to a data structure.

For all array-based queues our proposed warp optimizations are very similar. In order to implement warp optimization for the enqueue operations, we need to determine the number of threads that concurrently want to enqueue elements, assign a unique offset to each thread, and choose a lead thread to perform the operation. Using CUDA, the warp vote `__ballot(1)` will determine an active thread mask. The population count (`__popc`) of this mask yields the number of active threads. A combination of `__popc` with the value of the special low-level parallel thread execution (PTX) register `__lanemask_lt` yields a unique id for every thread. The thread with id zero then takes on the role of the lead thread. To communicate results, such as buffer offsets, one can use the shuffle (`__shfl`) instruction or resort to shared memory on older hardware.

Using warp optimizations for array queues not only has the advantage of greatly reducing the number of atomic operations that need to be executed, threads within a warp can also be guaranteed successive spots in the queue, which leads to much more efficient memory access patterns.

For the Collector Queue, a single thread can execute the atomic operation on the pointer, incrementing the pointer by the number of active threads and communicating the position of the first queue element to all threads. For the Mutex Queue and Dual Mutex Queue, a single thread can acquire the lock, advance the back pointer, and release the lock after completion. For the CAS Ordered Queue, the lead thread advances the back pointer, communicates the enqueue positions, waits for all threads to complete their memory transactions and then updates the ready pointer. For the Distributed Locks Queue and Pointed Queue, a single thread can manage the fill-rate counter and determine the spot of all elements. For the Local Queue, the atomic counter can be managed by the lead thread.

Link-based queues do not lend themselves to warp optimization as naturally as array-based queues. Nevertheless, some optimization is possible. When multiple threads concurrently enqueue elements in our Linked Queue, we at first let all participating threads allocate their nodes and write the data to their nodes. Then every thread communicates a pointer to its node to its neighbor. Knowing each neighbor's node, all nodes can locally be linked together forming a sub-queue. Afterwards, the lead thread simply has to exchange the global back pointer to point to the last element of the sub-queue and change the next pointer of the previous node at the back of the queue to point to the first element of the sub-queue. In this way, the elements of up to 32 threads can be inserted at once.

4.4.2 Combined Dequeue

Depending on the algorithm and execution framework, work items, item sets, and workpackages might be supported. If threads need to be dynamically assigned to the input element, barrier synchronization is needed. As the dequeue operation must be performed right before this barrier, the dequeue itself can also rely on synchronization without interfering with the execution framework. Thus, the dequeue requests of all threads within a block can be combined and a single thread can again perform the global operations. In case of an array based queue, this thread can alter the pointers to dequeue multiple elements at once. Each thread can then read its element, before the lead thread completes the dequeue operation.

4.4.3 Write Optimization

To make sure data becomes visible to other threads, one has to rely on volatile loads and stores to and from global memory. A naïveimplementation will likely compile to code accessing queue elements using scalar 4 or 8 byte loads and stores. Vector loads and stores would offer potentially much more efficient memory access for larger queue elements. To use vector loads and stores on volatile memory types, these commands must be manually supplied on current CUDA cards. Using inline PTX code suitable loads and writes can be achieved, leading to performance increases of up to 8 times. In case multiple threads are working together on a single element, *e.g.*, in shared memory, we can additionally distribute the required loads and stores amongst multiple threads, potentially decreasing the number of registers used by the kernel.

4.4.4 Distributed Storage Optimization

In certain situations, it is desirable to use multiple queues. For instance, if procedures work on item input, one would want to group items for each procedure in a separate queue. When using multiple array queues, the fill-levels of individual queues may vary greatly during runtime. Preallocating a buffer large enough to accommodate the maximum possible number of elements for each queue could be too wasteful. Instead, we suggest dynamically allocating the buffer memory for each queue from a shared pool of fixed size pages. This pool of pages can itself be managed by another queue, the *page queue*. In the beginning, all pages are unallocated and references to all pages are inserted in the page queue.

We replace the ring-buffers used in the standard queues with a small buffer of page references, which are then used to hold the queue element data, as shown in Figure 4.5. If a thread accesses the queue, it computes the page reference to be used based on the queue element it wants to access. It then checks if a page reference is present in this spot. If that is the case, it computes the offset on the page and accesses the according element

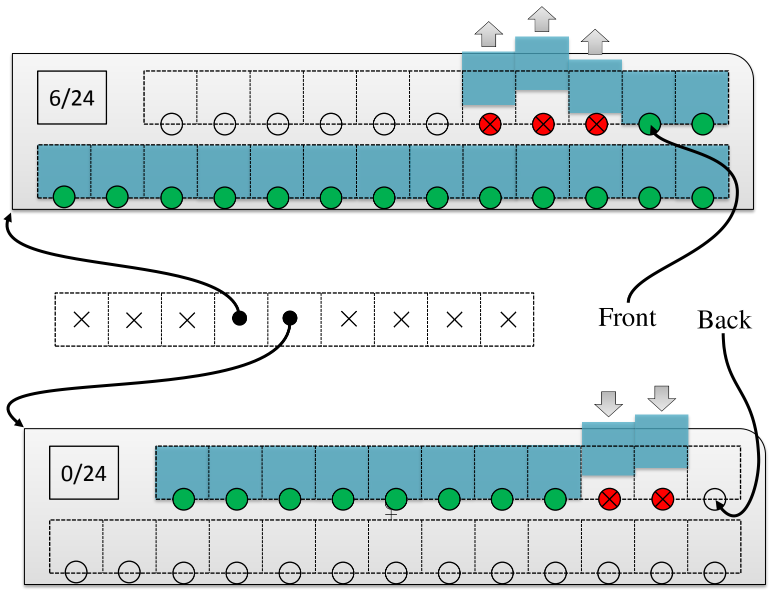


Figure 4.5: The distributed storage optimization uses external pages shared among different queues. Using a per page counter the current number of elements on each page can be tracked.

on the page itself. If no page reference is present, the thread draws a new page from the page queue and sets the page reference. To support freeing pages, we add an atomically operated counter to each page, which is increased during each dequeue. When a thread advances the counter to the number of elements that fit on one page, it was the last to remove an element from it and thus can safely free the page by unsetting the reference to it and inserting it into the page queue. The thread removing the last element from the page frees the page be unsetting the reference and inserting it into the page queue. For the page queue itself, we use one of the previously described queues.

If multiple threads concurrently try adding a page for the same reference, only the first one succeeds and the others put their pages back into the page queue. As many thousand threads could possibly concurrently try to add a page for the same reference, many pages would unnecessarily be removed from the page queue and put back. To avoid this situation, we only allow threads that insert an element into one of the first slots on the page to draw a new page, all others spin until one of the first ones sets the page reference.

4.5 Evaluation

The most relevant measures for queue performance are the average enqueue and dequeue times. To evaluate the performance of all queues, we tested them in two scenarios with varying number of threads. For each scenario, we start with a single block of 256 threads and increase the number of blocks until the device is fully occupied. In the first scenario, simulating an algorithm using workpackages, a single thread in each block enqueues one element, before all threads in the block cooperatively dequeue one element. In our tests, we use simple integers for the queue elements. For both, enqueue and dequeue, we measure the clock cycles needed. To get a reasonable estimate, we repeat this process 500 times within the same kernel. In the second scenario, simulating an algorithm that uses work items, every thread enqueues an element, before all threads in the block cooperatively dequeue the same number of elements. Again, we repeat the process 500 times.

For the queues that require a memory allocator, we do not use the rather slow memory allocator that ships as part of CUDA. Instead, we use a ring-buffer holding a fixed number of equally sized elements. Using one atomically operated counter, we request new elements from the ring buffer. As all the queues operate in a FIFO manner, elements that have been allocated first should also be freed first. Due to the high degree of parallelism, queues will not follow a strict FIFO ordering. As many queues rely on the memory allocator to avoid overflow situations, we additionally use an atomically operated bit mask to flag used elements. If the ring buffer wraps around and tries to allocate an element that has not been freed yet, we pretend to be out of memory instead of searching for an empty spot. An allocation requires only two global memory accesses, deallocation consists of a single atomic operation. While this memory allocator hardly deserves its name, we find it to be a reasonable model for the purpose of accounting for the overhead of dynamic memory allocation in our simulations.

To compare our queuing approaches to the state-of-the-art, we implemented the following non-blocking queues:

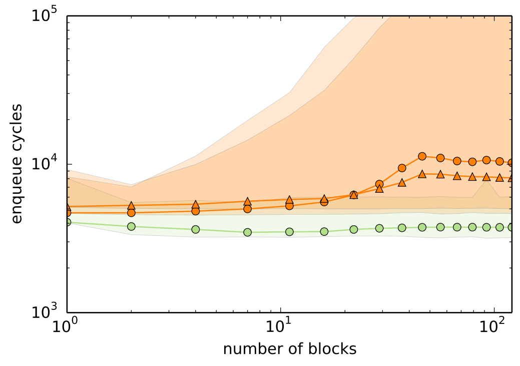
- Our implementation of the *Michael-Scott* queue [93] exactly follows their design. We use our memory allocator to offer a fair comparison. We also enhanced their queue with warp optimization similar to the warp optimization provided by our Linked Queue.

- Our implementation of the *Baskets Queue* [44], generates baskets as proposed in their original paper, inserting elements before the back of the queue if exchanging the back pointer failed. Again, we use our allocator and provide warp optimization.

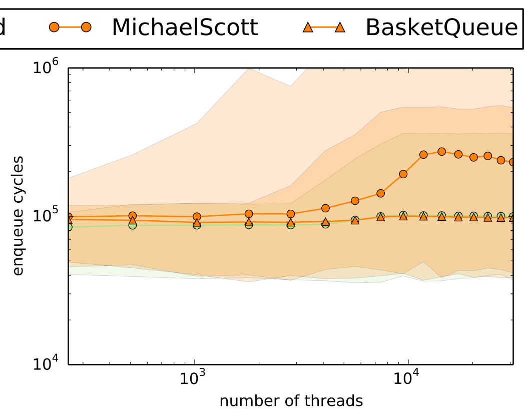
- Our implementation of the *Shann-Huang-Chen* queue [125], follows their algorithm. As elements are written to the underlying ring-buffer using atomic operations, we use our memory allocator to allocate the queue elements and only store pointers in the queue. We again provide warp optimizations for this queue, similarly to the warp optimization described for our array-based queues.

- Our implementation of the *Tsigas-Zhang* queue [140], suffers the same problems as the Shann-Huang-Chen Queue. A memory allocator is required, as individual elements need to inserted into the queue using atomic CAS operations. Their queue can be tuned to the ratio between the costs of atomic operations and read operations by altering a single parameter. We determined the best value for all tested GPU architectures.

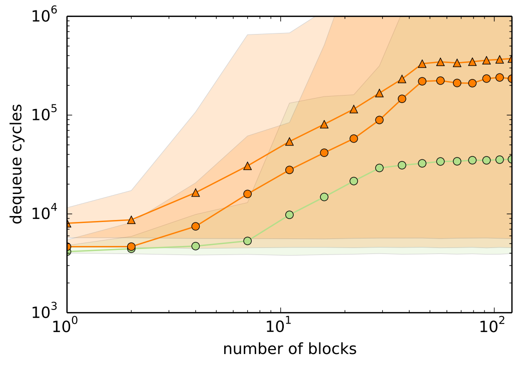
Figure 4.6, 4.7, and 4.8 show the average enqueue and dequeue cycles for our Linked Queue in comparison to the lock-free link-based Michael-Scott Queue and Baskets Queue. On the GTX 580, the Baskets Queue outperforms the Michael-Scott Queue for enqueue, in both the package scenario and the item scenario. For dequeue, the situation is reversed.



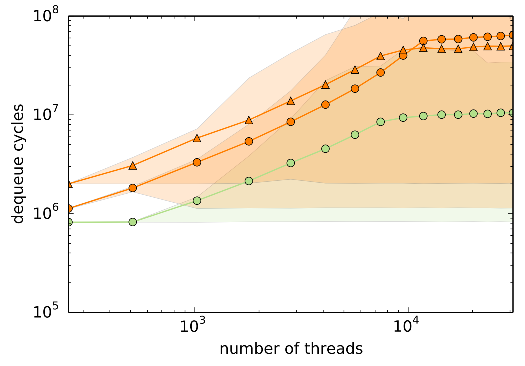
(a) package scenario, enqueue cycles



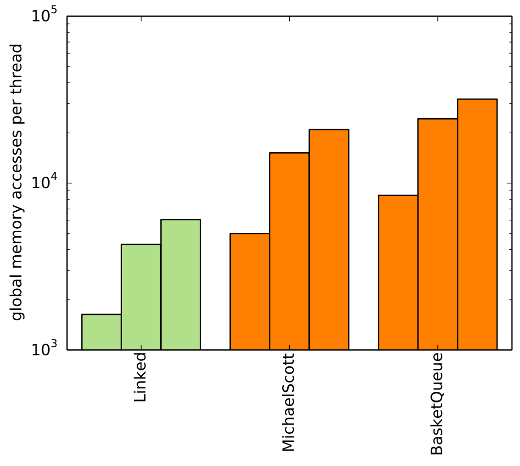
(b) item scenario, enqueue cycles



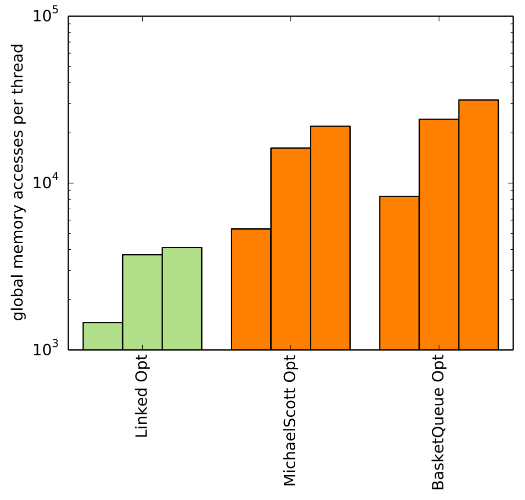
(c) package scenario, dequeue cycles



(d) item scenario, dequeue cycles

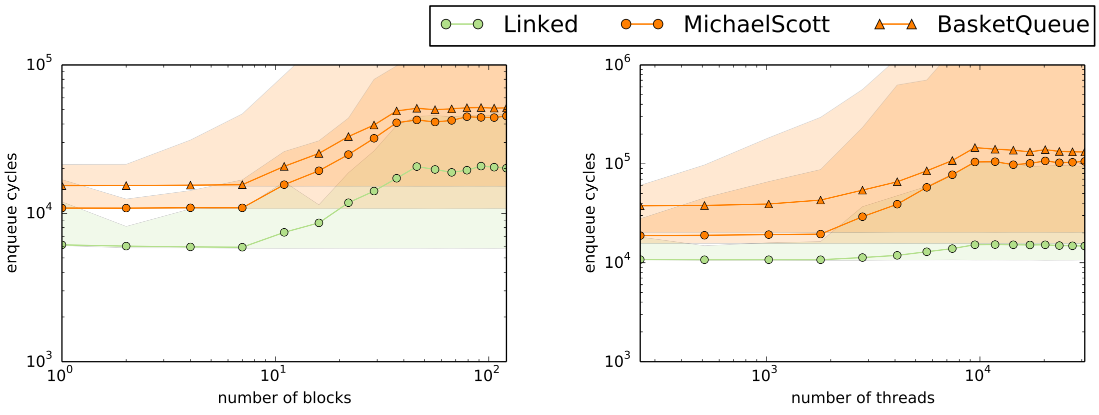


(e) package scenario, global operations



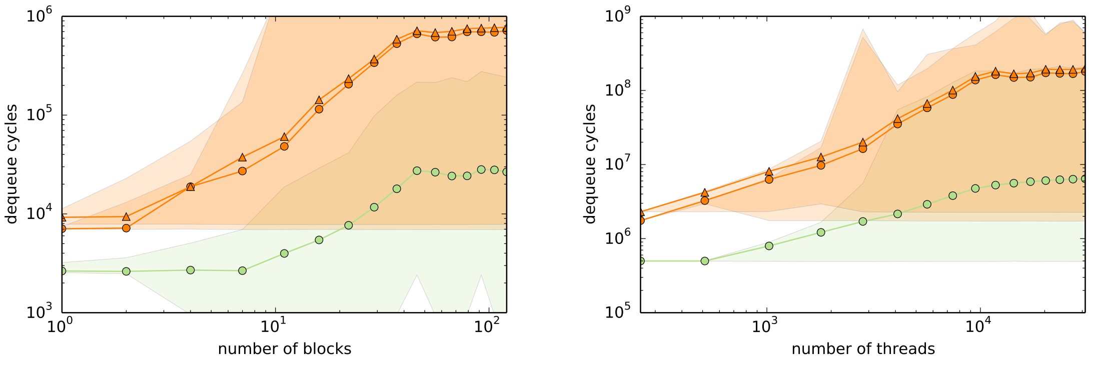
(f) item scenario, global operations

Figure 4.6: Performance comparison for the linked queues on a NVIDIA GTX 580. The number of global memory operations has been sampled at low, medium, and high thread counts. All queues use warp optimizations for the item scenario.



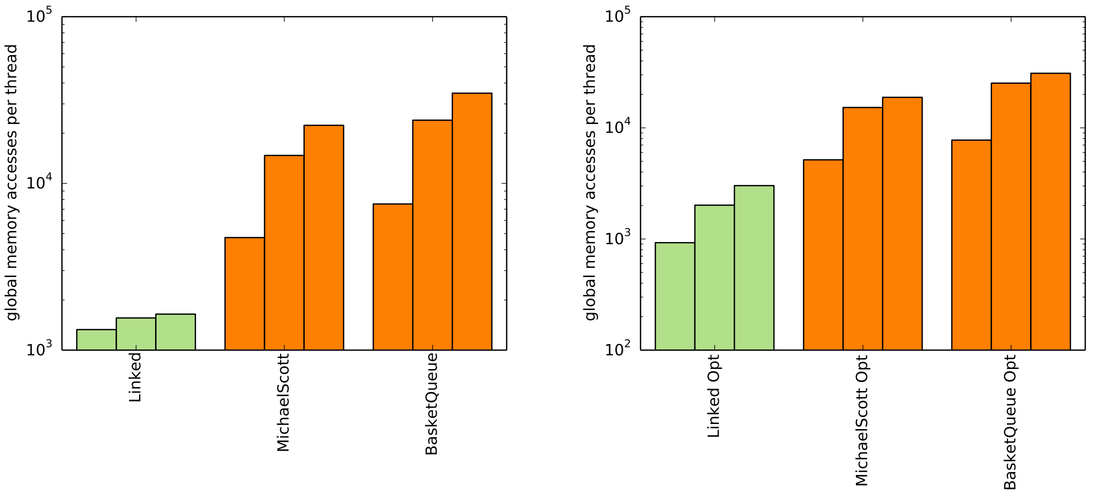
(a) package scenario, enqueue cycles

(b) item scenario, enqueue cycles



(c) package scenario, dequeue cycles

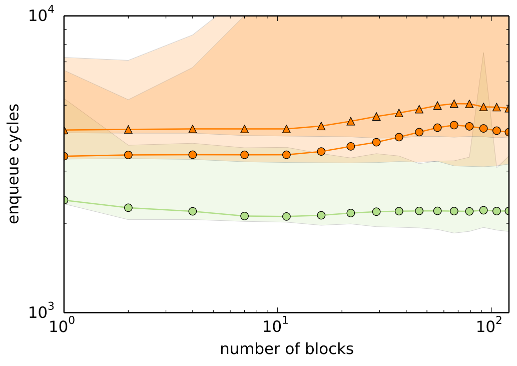
(d) item scenario, dequeue cycles



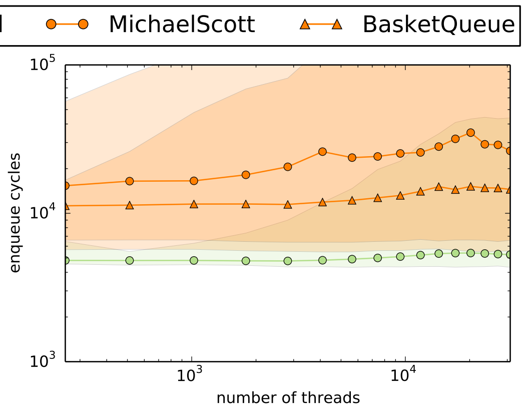
(e) package scenario, global operations

(f) item scenario, global operations

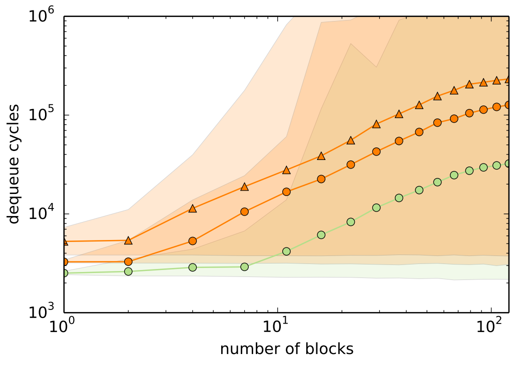
Figure 4.7: Performance comparison for the linked queues on a NVIDIA GTX 680. The number of global memory operations has been sampled at low, medium, and high thread counts. All queues use warp optimizations for the item scenario.



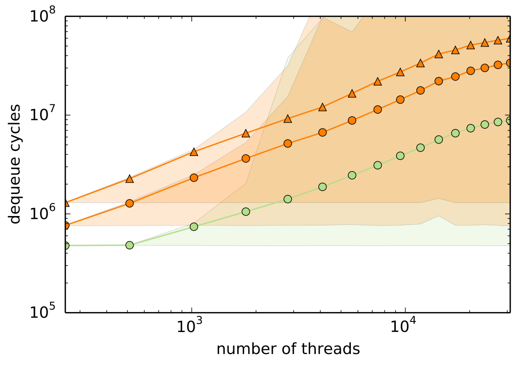
(a) package scenario, enqueue cycles



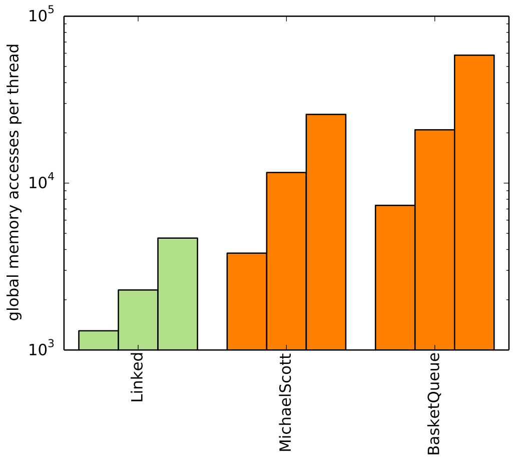
(b) item scenario, enqueue cycles



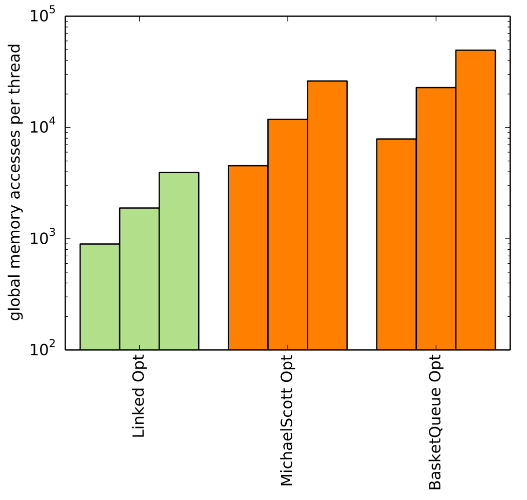
(c) package scenario, dequeue cycles



(d) item scenario, dequeue cycles



(e) package scenario, global operations



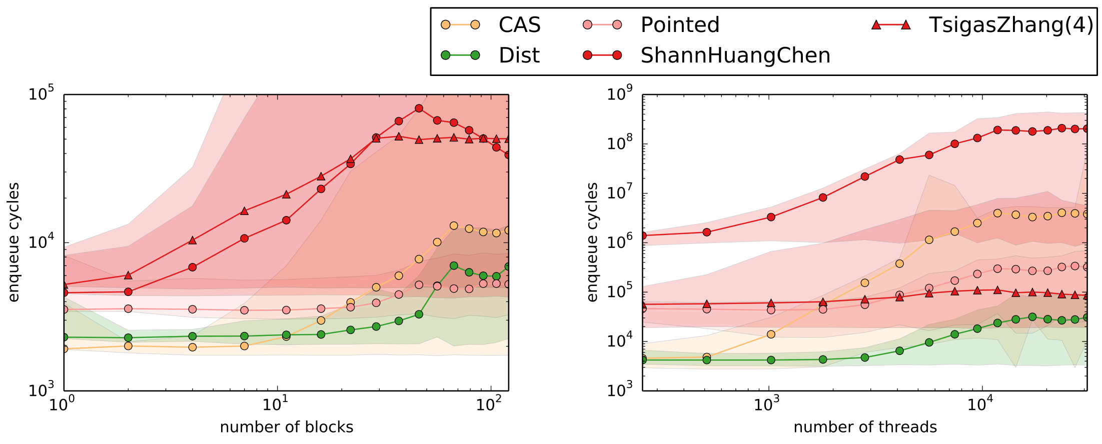
(f) item scenario, global operations

Figure 4.8: Performance comparison for the linked queues on a NVIDIA GTX TITAN. The number of global memory operations has been sampled at low, medium, and high thread counts. All queues use warp optimizations for the item scenario.

On the GTX 680,the Michael-Scott Queue outperforms the Baskets Queue in all respects. On the GTX TITAN, the situation is similar. Only for enqueue in the item scenario, the two queues switch position at approximately 1000 threads. The non-blocking link-based queues were significantly outperformed by our Linked Queue in all respects across all devices tested. For the item scenario, the difference is on the order of one magnitude on all devices. The advantage of our queue can not only be seen in the difference of enqueue and dequeue cycles, the number of global memory accesses also indicates that the non-blocking algorithms require a higher number of retries to complete their operations. Note that our queue also requires retries, but only during dequeue, while the Michael-Scott Queue and the Baskets Queue perform retries during enqueue and dequeue.

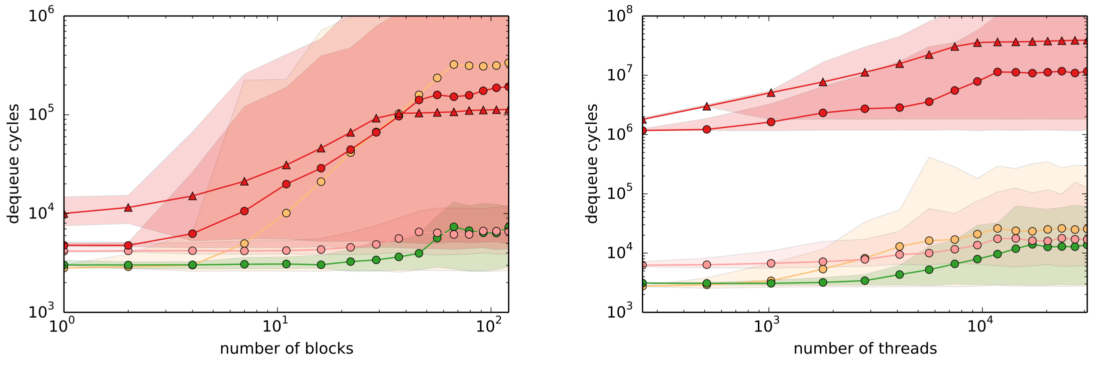
Figure 4.9, 4.10, and 4.11 show the average enqueue and dequeue cycles for the array-based queues. On all devices, our Distributed Locks Queue delivers the best performance in all test cases. The Pointed Queue is only slightly slower than the Distributed Locks Queue when compared to the other queues. The non-blocking queues are up to three orders of magnitude slower than our Distributed Locks Queue. Comparing the Tsigas-Zhang Queue with the Shann-Huang-Chen Queue shows that the Shann-Huang-Chen Queue is always faster than the Tsigas-Zhang Queue for the package scenario and for dequeue operations in the item scenario. The Tsigas-Zhang Queue achieved considerably better results for the enqueue in the item scenario. The enqueue operation for the item scenario is the most demanding operation throughout the entire test. It seems that the optimizations implemented by the Tsigas-Zhang Queue are only fruitful on the GPU if the number of threads concurrently accessing the queue becomes very high. The Dual Mutex Queue's performance is mostly comparable to the performance of the non-blocking queues. However, the dequeue operation in the item scenario is about as fast as the Distributed Locks Queue, which can be explained by the combined dequeue optimization. As the number of threads locking the queue during dequeue is rather low in comparison to the enqueue case, the second lock of Dual Mutex Queue is subject to little congestion. If the lock is acquired, dequeue is very efficient, as every thread can directly read its element. The Tsigas-Zhang Queue and the Shann-Huang-Chen Queue require more complex strategies to ensure consistency in their state variables. Even though the Dual Mutex Queue secures its data structures with only two locks, its performance is in the range of the non-blocking algorithms or even better.

On the GTX 580, the CAS Ordered Queue performs similar to the Dual Mutex Queue. On the GTX 680 and GTX TITAN, the CAS Ordered Queue is approximately a factor of two faster than the Dual Mutex Queue. On the GTX TITAN, the Distributed Locks Queue hardly shows any dependence on the thread count, it can serve enqueue and dequeue requests in approximately constant time. This behavior is only possible if the number of global memory operation per thread does not increase. The same behavior can be seen in the global memory operations. Again, the non-blocking algorithms require significantly more global memory operations than our blocking queues.



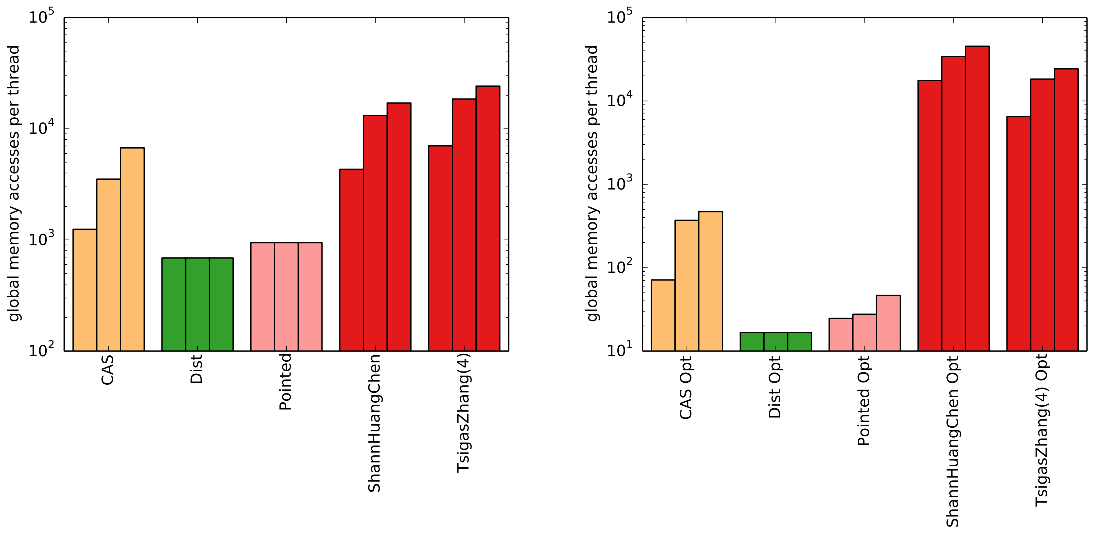
(a) package scenario, enqueue cycles

(b) item scenario, enqueue cycles



(c) package scenario, dequeue cycles

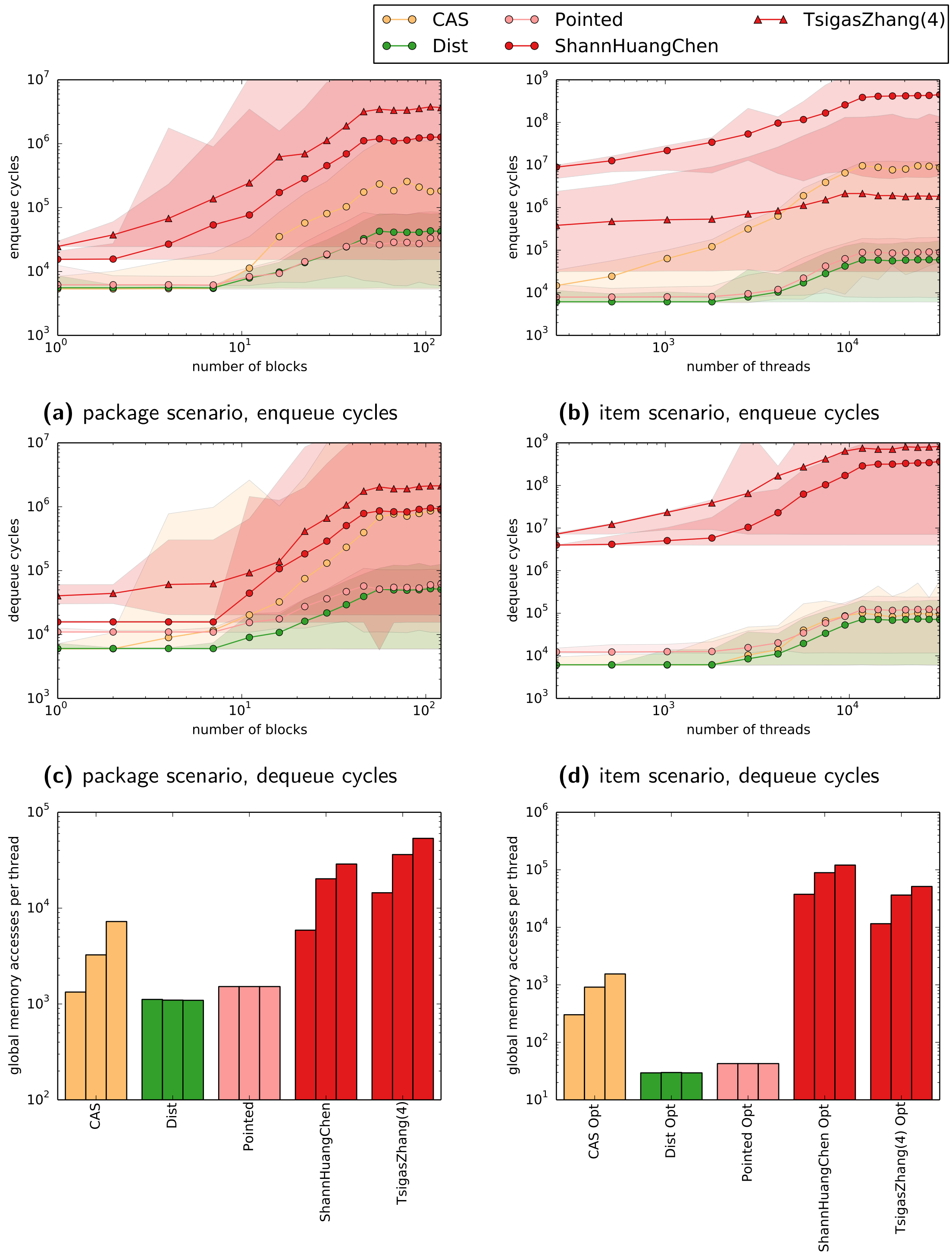
(d) item scenario, dequeue cycles



(e) package scenario, global operations

(f) item scenario, global operations

Figure 4.9: Performance comparison for the array-based queues on a NVIDIA GTX 580. The number of global memory operations has been sampled at low, medium, and high thread counts. All queues use warp optimizations for the item scenario. Note that the queues fill up the device at different thread counts.



(a) package scenario, enqueue cycles

(b) item scenario, enqueue cycles

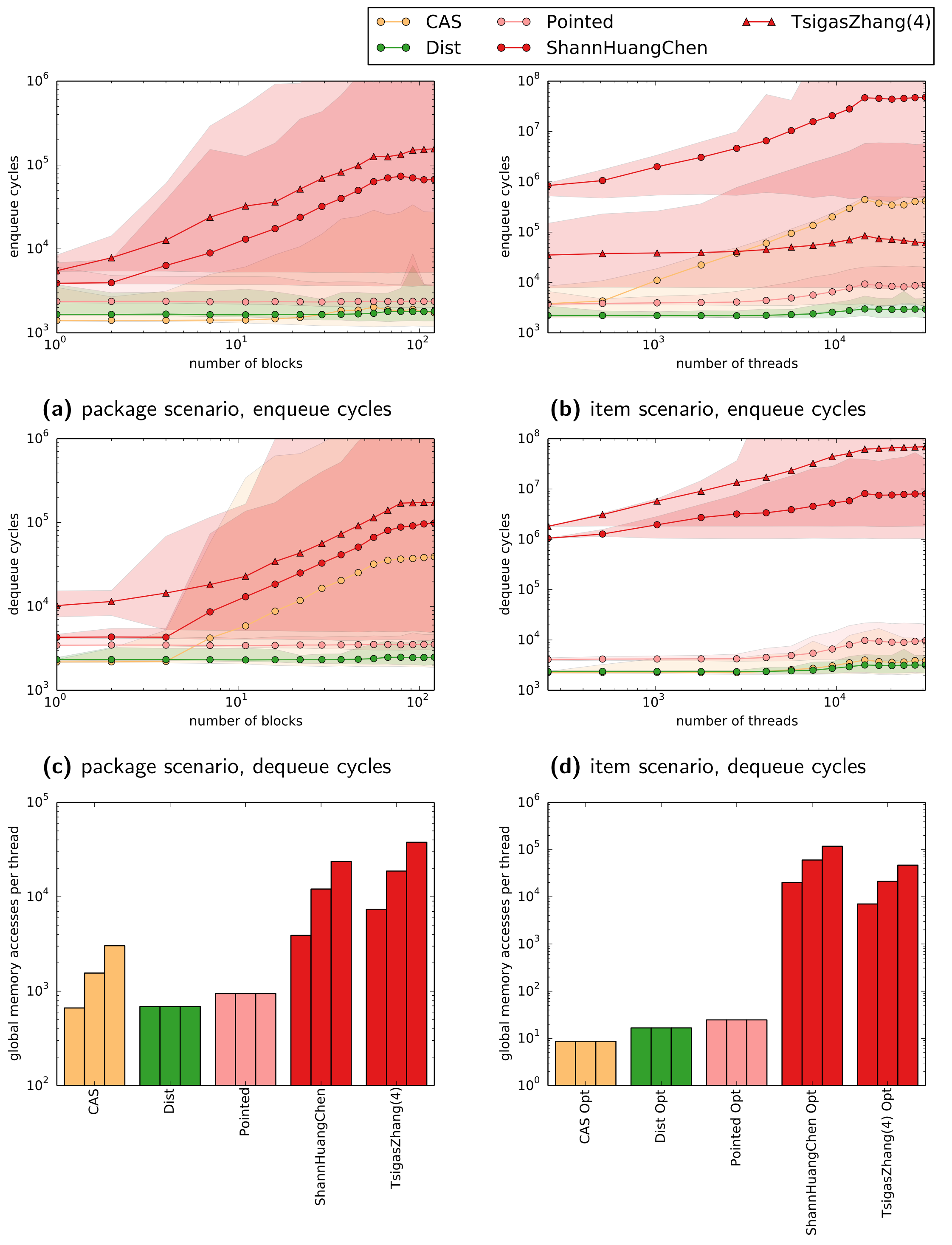
(c) package scenario, dequeue cycles

(d) item scenario, dequeue cycles

(e) package scenario, global operations

(f) item scenario, global operations

Figure 4.10: Performance comparison for the array-based queues on a NVIDIA GTX 680. The number of global memory operations has been sampled at low, medium, and high thread counts. All queues use warp optimizations for the item scenario. Note that the queues fill up the device at different thread counts.



(a) package scenario, enqueue cycles

(b) item scenario, enqueue cycles

(c) package scenario, dequeue cycles

(d) item scenario, dequeue cycles

(e) package scenario, global operations

(f) item scenario, global operations

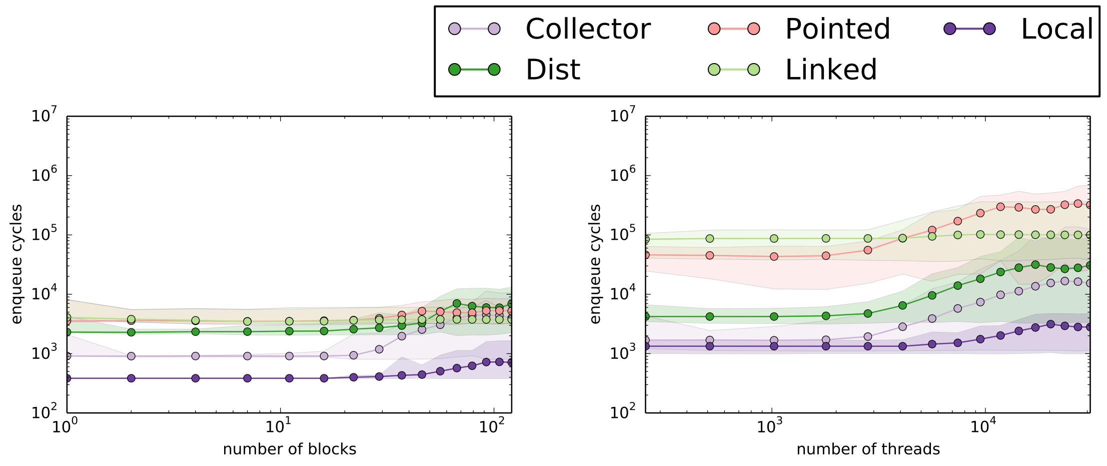
Figure 4.11: Performance comparison for the array-based queues on a NVIDIA GTX TITAN. The number of global memory operations has been sampled at low, medium, and high thread counts. All queues use warp optimizations for the item scenario.

To better judge the performance achievable by the global queues which are safe for autonomous scheduling on the GPU, we compare them with the Collector Queue and our Local Queue. These results can be seen in Figure 4.12, 4.12 and 4.14. On the GTX 580 and on the GTX TITAN, the Distributed Locks Queue is only twice as slow as the unsafe Collector Queue. The Pointed Queue is approximately five times slower. On the GTX 680, the dequeue operations show a bigger spread, the enqueue operations, however, show very similar performance for all global queues. The Linked Queue achieves competitive performance for the enqueue operations on all devices in both scenarios. Its dequeue performance, however, falls short from the performance of the other queues. This fact is also documented by the number of global memory operations needed by the queue, which are approximately 100 times more than the accesses required by the other queues.

On all devices, the Local Queue clearly outperformed all global queues. Due to the fast access to shared memory, the Local Queue is also significantly faster than the Collector Queue. The Local Queue was equally fast on all devices. For the global queues, there is a huge difference between the GPU architectures. In terms of cycles, the newer GTX TITAN with its improved atomic operations is between 2 to 20 times faster than the GTX 680. Also, the performance of the queues offering concurrent access to different items hardly decreases with increased thread counts on the GTX TITAN.

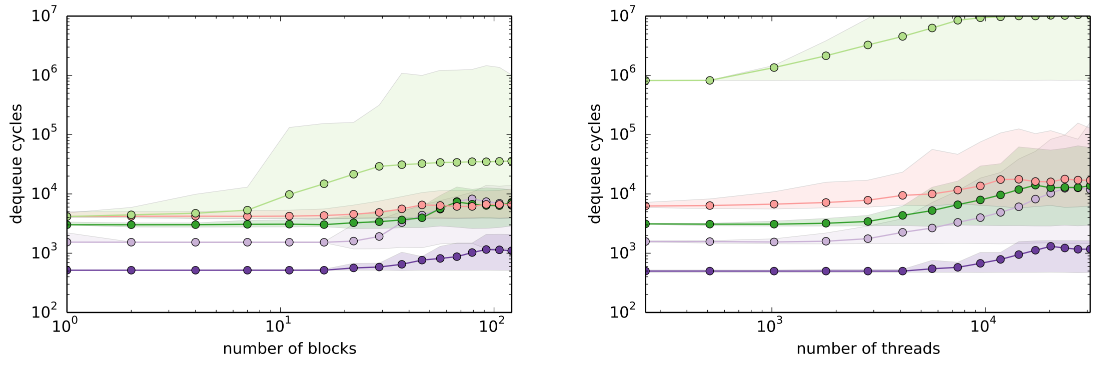
Table 4.2 shows the average enqueue and dequeue cycles for a fully utilized device for both scenarios on the GTX 580, 680, and TITAN. The tables clearly show that all three architectures have different implementations of atomic operations and thus warp optimization also has a different impact. In the case where only a single thread in each block performs an enqueue operation (package scenario), we see the small overhead of warp optimization. For the item scenario, the advantage of warp optimizations becomes apparent, leading to speed-ups of up to ten times for the faster queues and multiple orders of magnitude for the Mutex Queue and CAS Ordered Queue. Interestingly, warp optimizations have a very good impact on performance on the GTX 580, while their impact on the GTX 680 is rather small, which leads to the GTX 580 outperforming the GTX 680 with warp optimizations activated. The GTX TITAN shows a similarly good response to warp optimization as the 580. Using the distributed storage optimization (DestE) does not have a severe impact on performance. The local queues show a similar performance an all devices. The Dynamic Local Queue stays within a factor of two compared to the Local Queue. While this performance is sufficient on the GTX 580 and GTX 680 to be significantly faster than the Collector Queue, the efficient atomic operations on the GTX TITAN let the Collector Queue achieve results comparable to the Dynamic Local Queue.

In addition to the enqueue and dequeue times, we have summarized the most interesting properties of the queues in Table 4.2. The Collector Queue and both local queues only work correctly if all threads accessing the queue are synchronized between enqueue and dequeue operations. As the Collector Queue is a global queue, multiple kernel launches are needed to achieve this synchronization. Thus, it cannot be used as a work queue in dynamic load balancing approaches. While most queues only require a constant amount of additional memory, the Distributed Locks Queue, Pointed Queue and Linked Queue require additional memory for each queue element. The number of additional global memory transactions for enqueuing elements (for reasonable fill-levels) is constant for all queues except the Mutex Queue and the CAS Ordered Queue. While the number of locking



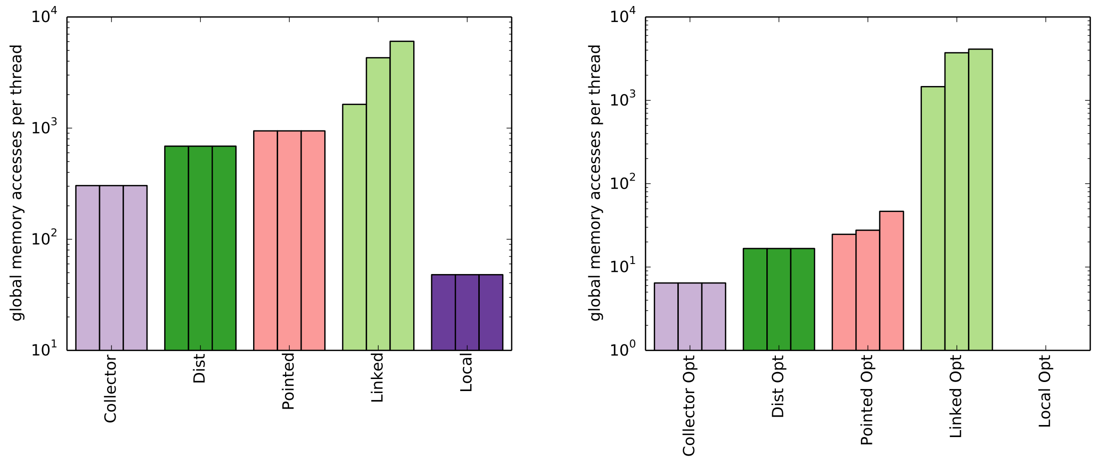
(a) package scenario, enqueue cycles

(b) item scenario, enqueue cycles



(c) package scenario, dequeue cycles

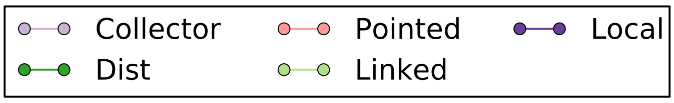
(d) item scenario, dequeue cycles

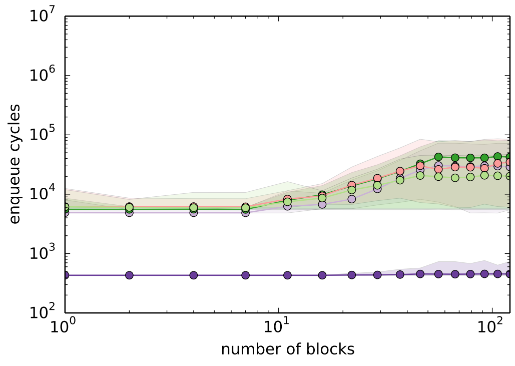


(e) package scenario, global operations

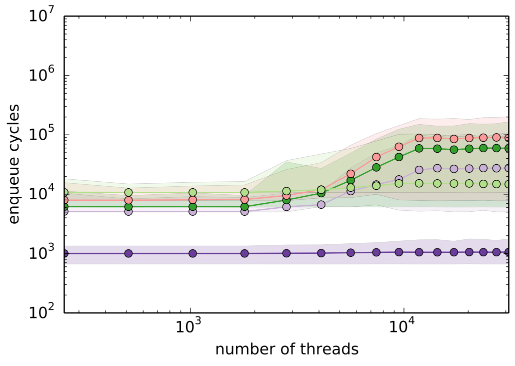
(f) item scenario, global operations

Figure 4.12: Performance comparison for the best queues on a NVIDIA GTX 580. The number of global memory operations has been sampled at low, medium, and high thread counts. All queues use warp optimizations for the item scenario. Note that the queues fill up the device at different thread counts.

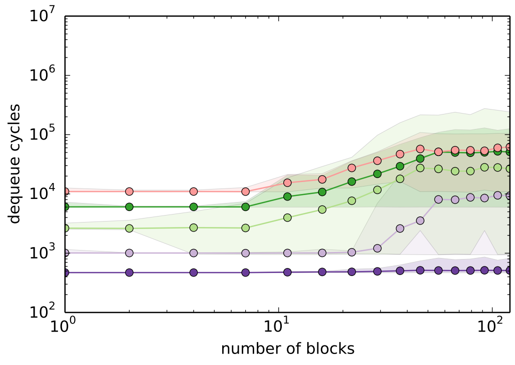




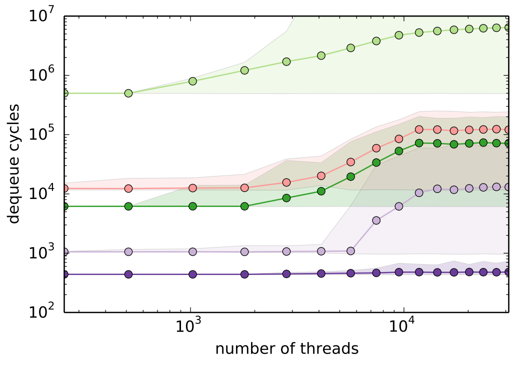
(a) package scenario, enqueue cycles



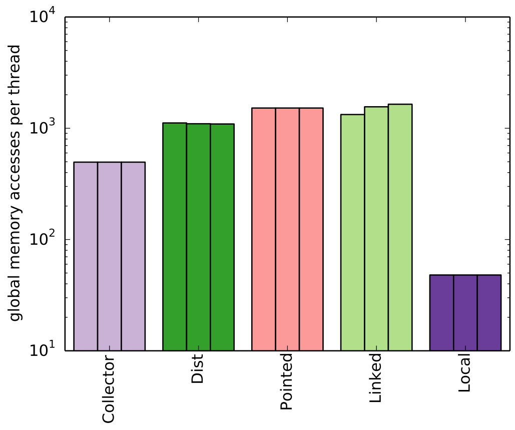
(b) item scenario, enqueue cycles



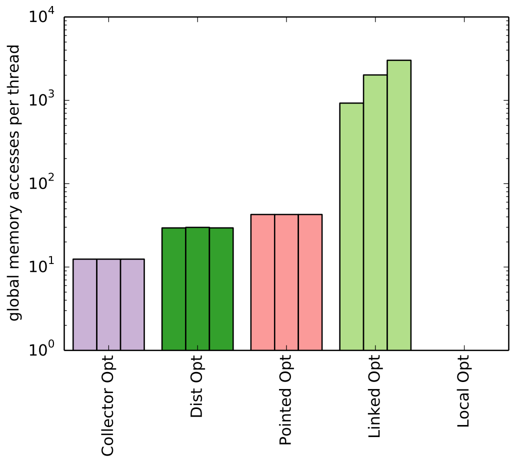
(c) package scenario, dequeue cycles



(d) item scenario, dequeue cycles

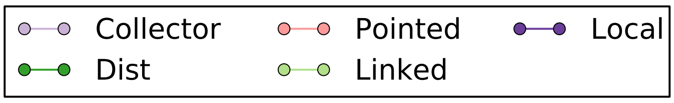


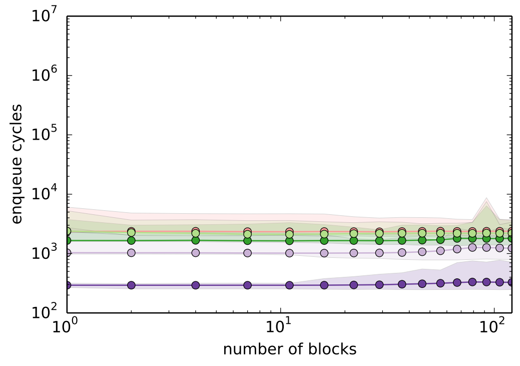
(e) package scenario, global operations



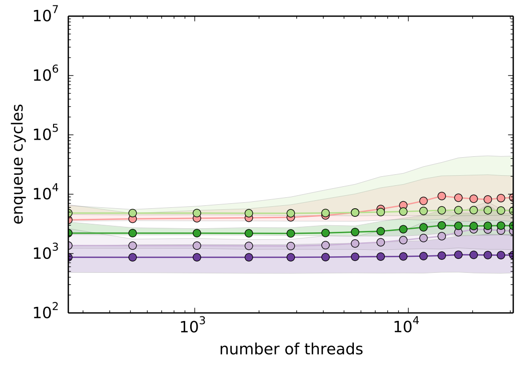
(f) item scenario, global operations

Figure 4.13: Performance comparison for the best queues on a NVIDIA GTX 680. The number of global memory operations has been sampled at low, medium, and high thread counts. All queues use warp optimizations for the item scenario. Note that the queues fill up the device at different thread counts.

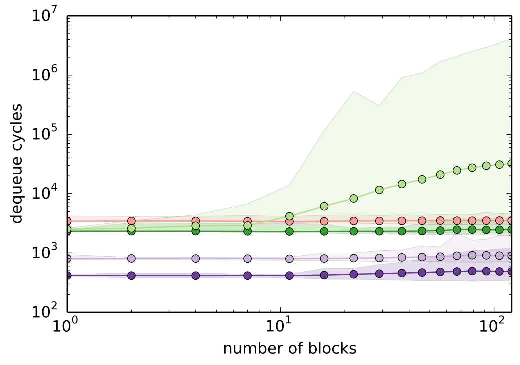




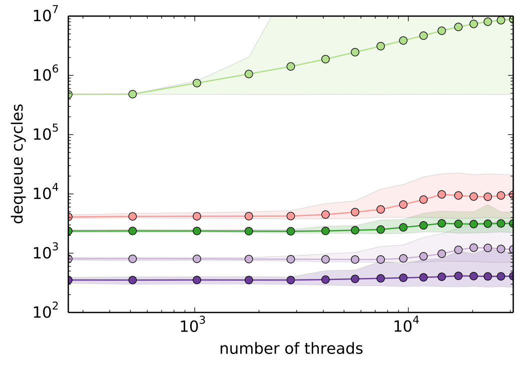
(a) package scenario, enqueue cycles



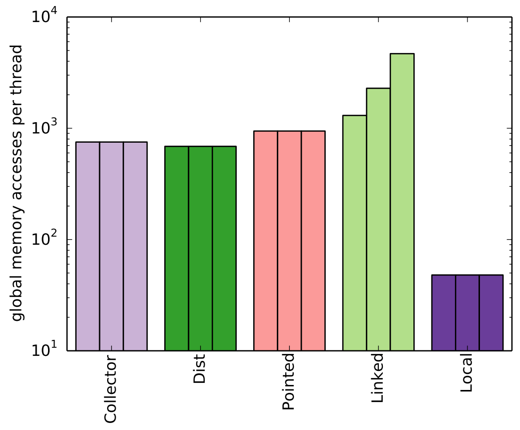
(b) item scenario, enqueue cycles



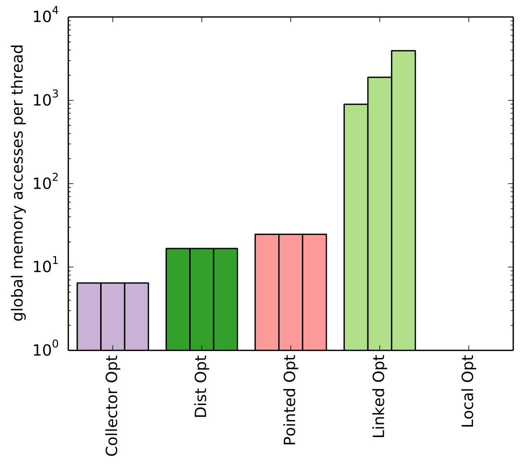
(c) package scenario, dequeue cycles



(d) item scenario, dequeue cycles



(e) package scenario, global operations



(f) item scenario, global operations

Figure 4.14: Performance comparison for the best queues on a NVIDIA GTX TITAN. The number of global memory operations has ve been sampled at low, medium, and high thread counts. All queues use warp optimizations for the item scenario.

Table 4.1: Average enqueue and dequeue cycles for the presented basic queues for both scenarios on a GTX 680, and GTX TITAN. All measurements were performed on fully utilized GPUs.

(a) package scenario, enqueue cycles

		Collector	DualMutex	CAS	Dist	DistE	Point	Link	Local	DLocal
580	std	3.07k	202k	10.1k	5.13k	4.37k	5.08k	3.77k	505.0	944.1
	opt	3.07k	251k	10.0k	4.97k	4.74k	5.48k	8.28k	1.09k	2.19k
680	std	30.4k	1.55M	234k	42.6k	42.8k	26.2k	19.7k	451.3	768.7
	opt	31.8k	2.99M	155k	45.0k	44.6k	30.3k	21.1k	612.7	1.04k
TITAN	std	1.11k	9.09k	2.02k	1.71k	2.31k	2.38k	2.20k	316.3	555.4
	opt	1.19k	17.8k	2.53k	1.87k	2.43k	2.56k	2.36k	516.8	754.4

(b) package scenario, dequeue cycles

		Collector	DualMutex	CAS	Dist	DistE	Point	Link	Local	DLocal
580	std	5.54k	83.1k	236k	5.65k	6.17k	6.38k	33.9k	815.6	2.07k
	opt	4.95k	28.2k	158k	5.22k	6.23k	6.55k	29.1k	753.4	1.97k
680	std	8.07k	3.53M	687k	51.2k	59.0k	51.3k	26.4k	511.1	1.34k
	opt	6.07k	1.86M	519k	50.9k	58.4k	55.7k	25.8k	514.3	1.30k
TITAN	std	864.0	125k	31.9k	2.39k	3.58k	3.52k	21.0k	477.6	1.21k
	opt	836.6	117k	29.8k	2.38k	3.60k	3.49k	20.7k	481.0	1.21k

(c) item scenario, enqueue cycles

		Collector	DualMutex	CAS	Dist	DistE	Point	Link	Local	DLocal
580	std	246k	1.31G	1.02G	586k	590k	588k	132k	21.6k	22.0k
	opt	11.3k	2.73M	3.69M	28.1k	32.7k	292k	101k	2.41k	4.01k
680	std	53.8k	1.56G	345M	78.0k	99.9k	103k	17.9k	16.2k	17.0k
	opt	27.5k	14.2M	8.81M	58.5k	81.8k	88.8k	15.2k	1.05k	1.40k
TITAN	std	13.3k	98.3M	197M	30.1k	21.4k	20.7k	6.04k	15.2k	16.4k
	opt	1.96k	504k	442k	2.98k	4.53k	9.32k	5.35k	924.2	1.15k

(d) item scenario, dequeue cycles

		Collector	DualMutex	CAS	Dist	DistE	Point	Link	Local	DLocal
580	std	8.35k	23.6k	141k	189k	199k	55.6k	10.0M	810.8	2.50k
	opt	6.07k	15.3k	23.8k	11.9k	16.5k	17.7k	10.0M	947.5	2.54k
680	std	6.48k	119k	111k	77.6k	98.9k	131k	6.07M	482.4	1.30k
	opt	12.2k	96.6k	90.4k	71.3k	91.6k	122k	5.60M	474.9	1.25k
TITAN	std	1.33k	6.37k	29.6k	6.89k	12.2k	13.4k	5.63M	377.0	1.24k
	opt	978.6	3.78k	4.00k	3.19k	4.94k	9.82k	5.66M	400.5	1.22k

Table 4.2: Basic queue overview: Only a subset of queues offer *concurrent* enqueue and dequeue and are thus usable for persistent threads approaches; two queues require a dynamic memory *allocator*; all but one queue support *reading and writing of multiple* elements in a single transaction; a few queues provide safe *access* to elements without removing them from the queue.
The *memory* requirements for three queues depend on the queue size n; in case of the Mutex Queue, the number of *operations for dequeue and enqueue* linearly depends on the threads c concurrently accessing the queue; for the CAS Ordered Queue and the Linked Queue, the number of operations depends on the order the memory transactions are executed (c^*); the number of *registers* needed by an enqueue or dequeue operation also varies.

	Collect	Mutex	CAS	Dist	Pointed	Linked	Local	DLocal
concurrent		●	●	●	●	●		
allocated					●	●		
multi r/w	●	●	●	●	●		●	●
accessible				●	●		●	●
memory	2	3	4	$3+n$	$3+n$	$2+n$	1	2
enqu. ops	1	$3+c$	$3+c^*$	4	3	3	1	2
dequ. ops	1	$3+c$	$3+c^*$	4	3	$3+c^*$	2	3
enqu. regs	11	12	17	12	14	9	6	8
dequ. regs	8	10	10	8	14	27	6	6

attempts increases linearly with the number of threads accessing the queue for the Mutex Queue, the number of required CAS operations for the CAS Ordered Queue depends on the scheduler and memory controller. If the memory transactions were processed in the same order as spots are assigned to threads accessing the CAS Ordered Queue, an enqueue operation would complete with no addition transactions. For dequeue, we see the same behavior for the Mutex Queue and the CAS Ordered Queue. The Linked Queue also requires more transactions if multiple threads want to dequeue elements concurrently, because all threads try to dequeue the same element. As only one thread can succeed, all other threads have to retry. In terms of register usage for enqueue and dequeue, all global queues behave pretty similarly ($8-14$ used registers). The CAS Ordered Queue forms an exception for enqueue (17 registers) and the Linked Queue needs 27 registers for dequeue. Shared queues outperform their global counterparts in all respects. They are not only faster in terms of access speed, they also only need a constant amount of additional memory and very few registers during enqueue and dequeue ($6-8$).

4.6 Discussion

Overall, we suggest to use local queues wherever possible, as they clearly outperform global queues in all respects. If a global queuing strategy is needed, we suggest using the Distributed Locks Queue, as it shows the best performance among all global queues that can be used for persistent threads approaches. Especially on the most current architecture, distributed locks outperform the other global queues in all tests. Interestingly, on this hardware, the Distributed Locks Queue can keep its performance almost constant across all thread counts, even if all threads on the entire GPU concurrently dequeue and enqueue items. It seems that, as long as warp optimizations are used, the problem of contention on single atomic counters is nearly solved on most recent hardware. The moderate amount of additional memory required by the Distributed Locks Queue is normally not a problem on current devices. If a large number of queues needs to be used, we recommend to employ the distributed storage optimization, as its impact on performance is not very severe. One of our most interesting findings is the fact that non-blocking queues perform significantly worse than our blocking algorithms throughout all tests. In some cases, our blocking queues were up to 100 times faster than the best non-blocking queue. Note that the source code for all our queues can be found in Appendix B and a performance plots for all testes queues are listed in Appendix C.

Chapter 5

Queuing Strategies

Contents

5.1 Load Balancing

While highly efficient queues are very important, the way queues are used to supply blocks with work is equally vital for an algorithm to be completed efficiently. The way queues store and give access to their elements defines the order of execution. In this way, the queuing strategy not only defines the priority of certain elements, it also influences the number of elements currently in the queue. If procedures generating many new elements are executed first, queues might grow large. If procedures that never generate new elements are scheduled, queue sizes will reduce. The internals of the queuing strategy, define where elements are stored, and how much pressure individual locks or counters will receive. Due to their influence on efficiency and load balancing, a well designed queuing strategy, thus, can boost the execution of the entire algorithm.

5.1.1 Monolithic Queuing

The probably most straight forward queuing strategy is to use a monolithic queue in global memory, as outlined in Figure 5.1. All worker blocks enqueue and dequeue elements with the same queue. Thus, scheduling is very consistent and *perfect load balancing* between different blocks is easily achievable. As only a single queue is used, the *risk of running out of memory is very low.* Only if the entire queue is filled, the queuing strategy can not serve further enqueue requests.

If different procedures are required for by an algorithm, different work elements get mixed in the queue. This fact entails a series of disadvantages. First, a monolithic queue *can only be used for work packages.* If items or item sets were mixed with package elements, utilization would ultimately drop as single items or item sets would be scheduled for execution. Second, *additional storage* is required for each element, to provide information about which procedure should be called for the element, increasing the overall memory requirement. Third, because elements for different procedures might be of different size, queues based on ring buffers might end up having a *high internal fragmentation.* In this

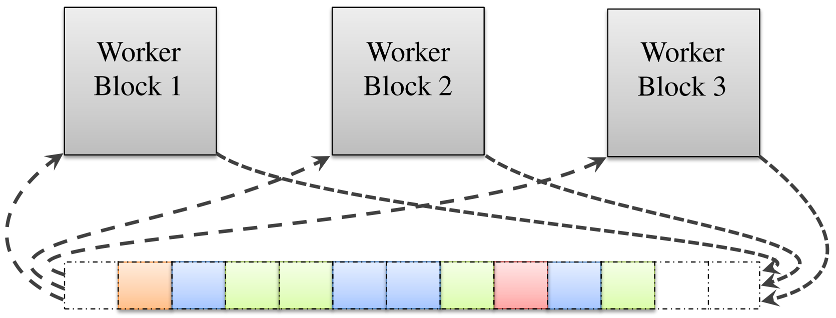


Figure 5.1: Monolithic queuing provides a single queue shared by all worker blocks.

case, every element in the queue must be able to hold the biggest element. Thus, for smaller elements, the remaining space is wasted.

In addition to these disadvantages, a single monolithic queue is a possible source for *very high congestion*, because all threads use the same lock or atomic counter when accessing the queue. A simple monolithic queue does not support *priorities* of any kind and thus cannot take any controllable influence on scheduling. As a result, queue lengths might grow very large or stay very low, as procedures with preferable input-output characteristics cannot be prioritized.

5.1.2 Per Procedure Queuing

The probably most intuitive queuing strategy for multi-procedure algorithms is to provide an individual queue for each procedure in global memory, as outlined in Figure 5.2. In this way, all blocks still access the same queues, providing *consistent scheduling and load balancing* among all blocks. Using a different queue for each procedure has a series of advantages: First, items, item sets, and packages are collected together, supporting *efficient execution of all granularities*. There is no need to store an addition identifier for each element, as *the queue itself provides information about the procedure to be executed. It is easy to prioritize one procedure over another,* which allows to react to non-optimal queue fill-rates. With knowledge about which elements can be generated during the execution of a single procedure, it is even possible to aim for homogeneous fill-rates among all procedures. The queuing strategy can select highly filled queues to reduce their fill-rate and avoid procedures generating elements for those queues.

There are only few disadvantages when using a queue per procedure in global memory. First, as all threads access the same queues, *high congestion might appear on single locks or atomic counters.* Second, *multiple queues need to be searched* to find elements suitable for execution, increasing the required memory bandwidth and steps until an element is found. If items and item sets are used, one might even search all queues twice. In a first run, elements are only dequeued, if there are sufficiently many elements to fill an entire worker block with elements. If this goal cannot be met, a second run is required. During the second run, any available element is taken out to advance the algorithm.

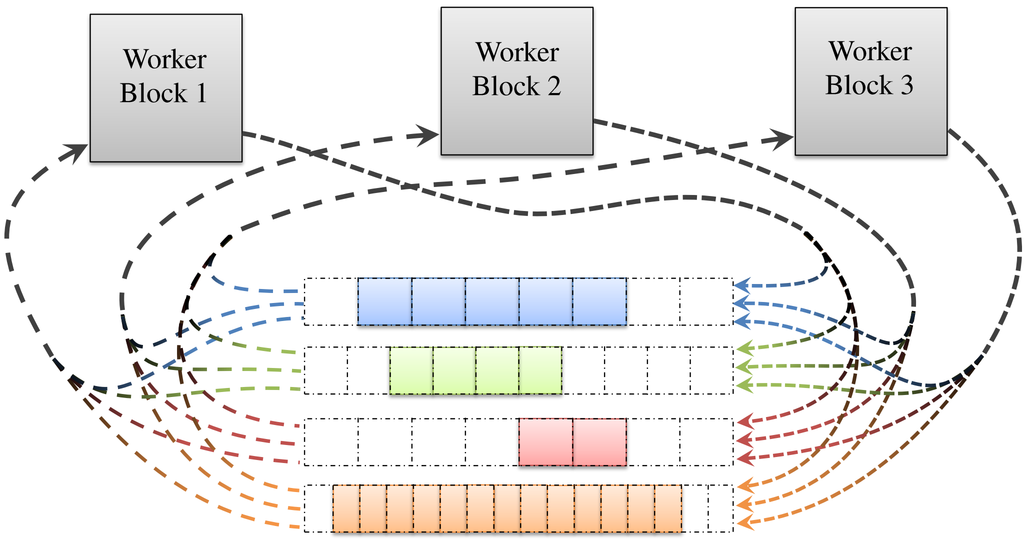


Figure 5.2: Per procedure queuing provides one queue for each procedure, avoiding the storage requirement for an additional identifier for each queue element.

5.1.3 Per Block Queuing

One strategy to reduce congestion on locks and atomic counters is to use more than a single queue, such as one for each worker block, as outlined in Figure 5.3. Per block queuing can also be combined with per procedure queuing, offering one queue for each procedure to each block. The big advantage of this setup is the *reduced number of accesses to the same locks and atomic counters*. The biggest disadvantage of the strategy is that there is *no shared scheduling strategy between different blocks*. Thus, load balancing between blocks is not supported per default. As the number of queues is increased, the overall *memory dedicated to queuing must also be increased* to be able to handle execution patterns, in which many elements are created by one block.

In order to introduce load balancing between blocks, two strategies have been proposed: *stealing* [10] and *donating* [141]. Both strategies involve accessing queues of other blocks.

Stealing works as follows: If a block's queue runs out of elements, it tries to steal elements from other blocks' queues. The order in which the other queues are chosen can be arbitrary, most often a simple round robin fashion is applied. Stealing has the advantage that *all blocks are supplied with work until the entire algorithm has finished*. One disadvantage is an increased number of queue accesses when running low on elements as all blocks possibly try to access all queues, which increases the memory bandwidth usage and delays the shut down phase. When using stealing, still the same amount of memory needs to be reserved for all queues, as individual queues might still fill-up quickly, while all other queues only hold few elements.

Donating works as follows: If a block's queue is running full, the block tries to enqueue new elements with another block's queue. If the chosen queue is also full, it tries to enqueue with another queue in a round robin fashion. In this way, other blocks can also be supplied with work. The advantage of this strategy is that the *queue size can be chosen smaller*,

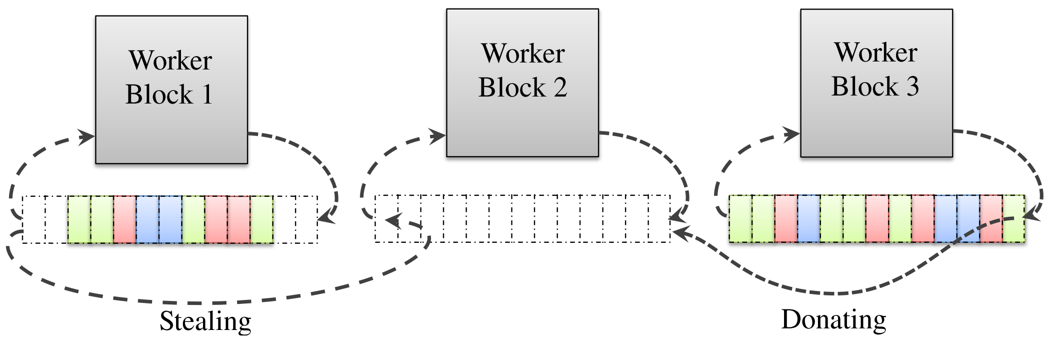


Figure 5.3: Per block queuing provides an individual queue for each worker block. To be able to distribute work between different queues, stealing and donating can be used. If a queue runs out of elements, it can steal elements from another block's queue. Before a queue overflows, elements can be donated to other queues which are still able to hold elements.

as elements are simply moved to another queue if one is running out of memory. One disadvantage is an increased number of queue accesses when the queues are about the run full. Another, even more severe problem is that the*success of the strategy depends on the queue size*. If queues are chosen too big, all but one block might stare for a serious amount of time. If queues are chosen too small, the execution might fail as there is not enough space to store all elements. Also, donating cannot be applied to every type of algorithm. If an algorithm is contracting only, *i.e.*, every procedure emits up to one new element, donating will never happen and, thus, no load balancing will apply.

5.1.4 Locally Buffered Queuing

While global queues are essential to share work between blocks, they require a round trip to relatively slow global memory. An alternative is to place queues in fast local shared memory. As shared memory is very limited in size, only small queues can be made available in shared memory and a backup by a global queue is most often required, as outlined in Figure 5.4. The advantage of locally buffered queuing is clearly the provided *access speed*. The disadvantage is that it *hinders global scheduling strategies*, as each block only has access to its own queue.

The variables that influence local queuing are the chosen queue size and the threshold determining when elements can be taken out of the queue. Similar to donating for per block queuing, a too large queue might cause starvation of other blocks. If queues are chosen too small, they might become inefficient, because too many elements need to go through global memory. If there are not enough elements available to fill up an entire block, it is most often a good idea to draw work from the global queue and wait until there is enough data locally available to provide work for all threads of the block. However, this bares the risk that the queue suddenly overflows and elements need to be moved to global memory. Thus, one might choose the threshold determining when elements should be taken out of the queue a bit below the work needed for all threads. The best choice for these variables again depends on the executed algorithms.

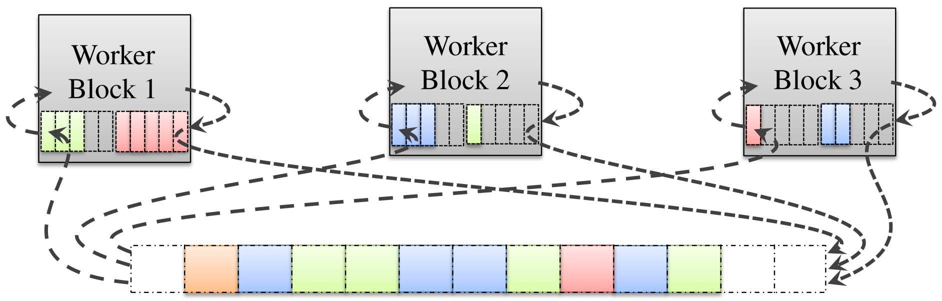


Figure 5.4: Locally buffered queuing provides small queues in fast on-chip shared memory. These small queues are backed by larger queues in global memory.

Another factor for local queuing is whether a queue for a certain procedure should be provided in shared memory. If an algorithm by itself already needs a large amount of shared memory, it might be beneficial to offer local queues only for selected procedures. According to our experiments, procedures which form the end of pipelines, take small items or item sets as input, or show rather short execution times are a good fit for local queuing.

5.1.5 Evaluation

To evaluate the different queuing strategies, we used a simple persistent threads approach. As test algorithm we use a single recursive procedure, either based on packages or based on work items. In both cases, we started a fixed number of initial packages or items and with a certain probability re-emitted them. For the first 30 iterations, we use a re-emit probability of 100%. For every following iteration, we reduce the probability by 3% leading to a maximum iteration count of 64. To simulate the time required for the execution of individual elements, we either perform 512 fused multiply-adds (FMA) to simulate arithmetic load or perform 64 memory transactions to simulate a memory bounded algorithm. In addition, we also vary the size of work items (16 and 64 bytes) and packages (16 and 256 bytes) to evaluate how the queuing strategies can cope with increasing memory requirements and data access times. To simulate different algorithm characteristics, we run the same test again, but instead of only re-emitting the same element, we emit a second element with a certain probability. We setup this probability to counterbalance the reduction of the re-emit probability, overall keeping approximately an equal number of elements in the system for the first 30 iterations. Then, we again reduced the number of emitted elements linearly. This second test forms a more challenging scenario for the load balancing strategies, as the number of created elements differs between different processors.

The results of the first set of tests are shown in Table 5.1, 5.2, and 5.3. Items are only supported for per procedure queuing. Comparing the basic global queue approaches, Distributed Locks Queues achieved the best performance. If only a single global queue is used, the Linked Queue, the Mutex Queue and the CAS Ordered Queue are clearly slower than the Distributed Locks Queue. When per-block queuing (especially with stealing) is applied, the slower basic queues can to a certain degree close this gap. The performance of simple per-block queuing seems to be limited by the missing load balancing strategies. The performance of stealing and donating is very similar in about 90% of all tests. In about ten percent of the cases, however, stealing outperformed donating by up to 50%. The problem with donating in this scenario is the fact that donating is only able to balance the load between the processors during the first stages, when all queues are fully filled. As soon as the number of executed elements is reduced, load inconsistencies can arise. Stealing is still able to handle these situations.

For item data the difference between the basic queues becomes even more severe. In the simple setup, the Distributed Locks Queue is between 30 and 300 times faster. When using per-block stealing, the difference is reduced to 2 to 50 times.

The most interesting point of this evaluation is that per-block queuing with the best load balancing strategies hardly outperforms queues that are shared between all processors if the Distributed Locks Queue is used as underlying queue. Especially on the GTX TITAN the difference between per-block stealing and using one shared queue is below 5%. In half of the cases, the one queue strategy even outperformed per-block stealing.

When comparing monolithic queuing with per procedure queuing, there is hardly any difference. The slightly decreased number of memory loads of per procedure queuing does not show up in the performance results.

A strong impact on performance can be achieved when using locally buffered queuing. On the GTX 580 the performance can be increased by up to 80 percent and locally buffered queues were the fastest in all tests. On the GTX 680 locally buffered queues were also leading the score board in all cases. Interestingly, the impact of locally buffered queuing was more than ten times stronger on the 680 than on the 580. On the 680 it boosted performance by a factor of up to 15, indicating that the queuing operations (including their memory accesses) otherwise form the bottleneck on the 680. On the GTX TITAN, the performance of locally buffered queuing and distributed locked global queues was very similar. In only two cases locally buffered queuing was faster than the global strategy. The amount of memory reserved for locally buffered queuing also has an influence on performance. In this scenario, the memory was used for queues, the faster the approach got. As soon as a sufficient threshold was reached, performance did not increase any further. The threshold was reached, when there was sufficient space for storing all elements in shared memory. For item data, up to 16kB were needed.

The results of the modified test with up to two elements being created in each procedure are shown in table 5.4, 5.5 and 5.6. In general, we see a similar picture. Distributed Locks Queues perform best among all global queues. Per-block stealing significantly can increase the performance of the slower queues. On the GTX TITAN per block stealing outperformed the a single Distributed Locks Queue only in 2 of 8 cases. On the GTX 680 and 580 this ratio was 3:8 and 5:8, respectively. These results indicate that the newer devices among other things offer more efficient atomic operations, which are the only source for congestion for the Distributed Locks Queue. Per-block donating lost ground in comparison to the previous test case compared to per-block stealing. Although the number of element in the queues only slightly varies for the first iterations, the remaining iterations seem to generate severe imbalances in the element distribution between the blocks.

Locally buffered queuing performed best in 4 of 8, 8 of 8, and 3 of 8 cases on the GTX 580, 680, and TITAN respectively. The difference between global and local queues was again big on the GTX 680. However, the success of locally buffered queuing overall

Table 5.1: Recursive tests - performance (GFLOPS and GB/s) on a GTX 580

			Packages				Items			
			FMA		Memory		FMA		Mem	
			Small	Big	Small	Big	Small	Big	Small	Big
Linked	Mono		314.1	308.5	75.2	76.5				
		PerBlock	250.2	247.5	68.0	67.0				
		Stealing	504.9	388.8	119.1	93.9				
		Donating	521.6	359.6	112.5	84.8				
	PerProc		319.8	310.7	78.4	74.8	1.3	1.2	0.3	0.3
		PerBlock	254.5	246.0	66.9	65.4	6.4	5.4	1.6	1.3
		Stealing	513.6	401.0	119.0	97.9	6.7	5.2	1.7	1.3
		Donating	559.3	366.0	112.0	86.7	6.3	4.9	1.6	1.2
Mutex	Mono		67.6	69.3	17.3	17.2				
		PerBlock	399.4	347.8	95.1	84.2				
		Stealing	748.6	666.7	122.1	116.5				
		Donating	678.4	642.1	117.7	117.8				
	PerProc		83.2	73.8	20.3	18.6	5.7	7.9	1.4	2.0
		PerBlock	384.1	317.5	90.6	84.5	171.0	115.1	44.7	30.3
		Stealing	718.4	563.5	121.6	107.5	172.6	114.6	44.1	31.4
		Donating	654.7	554.6	117.0	115.2	158.0	114.8	42.2	29.9
CAS	Mono		148.7	148.5	35.9	35.8				
		PerBlock	387.5	343.8	92.9	78.2				
		Stealing	752.0	583.5	122.9	109.8				
		Donating	731.6	616.7	118.1	113.9				
	PerProc		131.2	132.3	30.8	31.7	10.6	17.8	2.8	4.3
		PerBlock	394.0	316.7	83.3	71.6	352.4	202.8	88.6	54.9
		Stealing	701.6	512.2	121.0	98.8	370.3	199.8	94.6	53.3
		Donating	693.6	596.6	118.4	112.1	358.1	201.7	93.3	55.1
Dist	Mono		749.3	685.6	124.2	117.8				
		PerBlock	374.3	351.4	89.1	70.5				
		Stealing	765.1	674.5	124.0	116.0				
		Donating	750.4	611.6	118.9	112.9				
	PerProc		685.0	608.3	122.3	117.7	408.8	223.2	96.0	59.4
		PerBlock	371.5	304.7	85.4	72.5	456.6	211.5	101.1	57.6
		Stealing	823.2	599.0	122.9	117.5	487.7	227.2	108.6	61.6
		Donating	661.3	616.9	118.3	116.1	447.6	209.8	101.9	56.9
Local		Small	816.4	580.5	120.5	113.8	312.6	197.9	78.5	55.3
		Medium	**854.4**	700.0	**125.2**	120.2	455.5	217.2	99.9	60.0
		Large	853.0	699.5	125.1	**120.3**	542.3	242.3	108.8	66.0
		Huge	851.4	**702.6**	125.1	119.6	**569.9**	**280.8**	**117.9**	**74.1**

Table 5.2: Recursive tests - performance (GFLOPS and GB/s) on a GTX 680

			Packages				Items			
			FMA		Memory		FMA		Mem	
			Small	Big	Small	Big	Small	Big	Small	Big
Linked	Mono		247.8	259.8	54.1	58.0				
		PerBlock	158.8	155.7	29.8	31.3				
		Stealing	303.2	253.6	64.3	57.7				
		Donating	260.2	224.6	55.1	49.8				
	PerProc		247.9	248.0	54.2	56.3	1.7	1.6	0.4	0.4
		PerBlock	153.2	147.4	34.4	33.5	7.1	5.5	1.7	1.3
		Stealing	294.9	261.5	65.3	58.0	7.4	5.6	1.8	1.4
		Donating	219.7	223.3	48.4	51.5	7.1	5.5	1.7	1.4
Mutex	Mono		2.3	6.0	0.5	1.4				
		PerBlock	16.5	40.2	3.7	9.8				
		Stealing	51.5	84.8	13.1	20.4				
		Donating	31.3	83.4	7.5	20.1				
	PerProc		3.7	5.5	0.9	1.3	0.4	0.7	0.1	0.2
		PerBlock	37.3	54.1	8.9	13.5	4.2	6.7	1.1	1.6
		Stealing	87.1	85.3	22.7	20.0	6.4	6.6	1.7	1.6
		Donating	33.5	84.0	8.2	20.8	2.8	6.7	0.7	1.6
CAS	Mono		16.8	31.4	4.1	7.7				
		PerBlock	80.0	95.1	19.4	20.8				
		Stealing	136.8	169.2	32.1	39.2				
		Donating	98.6	160.9	23.3	38.2				
	PerProc		17.1	31.4	4.1	7.6	1.3	2.7	0.3	0.6
		PerBlock	75.5	101.4	20.2	21.4	17.2	24.8	4.2	6.1
		Stealing	176.8	155.9	42.0	37.2	29.5	24.6	6.9	6.0
		Donating	98.2	159.0	23.3	38.7	12.9	18.9	3.1	4.7
Dist	Mono		112.1	181.9	26.4	43.4				
		PerBlock	65.2	106.9	15.5	22.9				
		Stealing	141.7	170.2	35.4	39.4				
		Donating	104.8	163.7	25.3	40.7				
	PerProc		110.8	182.7	26.6	46.5	81.3	48.6	20.1	11.5
		PerBlock	67.2	101.4	16.5	27.8	77.1	47.9	17.6	11.4
		Stealing	139.7	169.8	25.8	44.1	74.7	49.8	17.9	12.0
		Donating	106.2	169.1	24.9	45.5	49.8	34.6	11.0	7.6
Local		Small	467.7	368.3	99.0	80.9	131.0	70.3	30.2	17.1
		Medium	**1105**	836.0	133.6	134.2	133.6	77.3	30.8	18.1
		Large	1103	841.9	**133.7**	**134.4**	242.0	86.7	61.7	19.5
		Huge	1104	**841.9**	133.6	134.3	**728.2**	**108.2**	**164.0**	**23.4**

Table 5.3: Recursive tests - performance (GFLOPS and GB/s) on a GTX TITAN

			Packages				Items			
			FMA		Memory		FMA		Mem	
			Small	Big	Small	Big	Small	Big	Small	Big
Linked	Mono		409.6	399.0	99.8	98.1				
		PerBlock	638.0	488.3	151.8	94.2				
		Stealing	1065	742.1	170.9	162.5				
		Donating	810.1	613.4	157.1	115.9				
	PerProc		415.7	399.8	102.9	97.6	1.7	1.6	0.4	0.4
		PerBlock	736.6	466.5	118.6	115.9	7.8	5.9	1.9	1.5
		Stealing	1168	890.9	147.1	163.4	12.8	9.9	3.1	2.5
		Donating	917.0	716.5	129.9	137.4	7.7	5.7	1.7	1.4
Mutex	Mono		88.9	72.4	20.5	17.1				
		PerBlock	470.6	417.1	109.7	89.5				
		Stealing	809.5	857.9	159.3	159.5				
		Donating	841.9	672.5	164.1	128.8				
	PerProc		92.9	81.0	20.9	19.4	10.8	9.8	2.7	2.5
		PerBlock	478.7	352.7	98.3	90.1	116.4	89.8	28.2	22.1
		Stealing	873.3	753.2	136.0	149.2	111.4	94.0	28.3	23.5
		Donating	918.6	782.8	137.9	149.6	116.3	92.3	27.5	22.8
CAS	Mono		344.2	337.3	83.6	82.3				
		PerBlock	558.1	427.0	124.8	103.7				
		Stealing	972.7	1000	172.8	167.9				
		Donating	1045	755.7	178.9	134.8				
	PerProc		335.1	339.2	81.4	82.1	36.0	52.2	8.9	13.1
		PerBlock	572.8	446.6	116.6	104.7	351.4	193.8	79.9	48.0
		Stealing	1100	998.0	144.2	169.1	487.9	285.5	109.5	67.1
		Donating	1033	882.7	148.2	157.9	359.7	209.2	81.4	51.2
Dist	Mono		1195	1059	**185.3**	168.3				
		PerBlock	550.0	490.7	116.5	106.1				
		Stealing	1036	1075	179.6	169.3				
		Donating	1031	751.4	180.4	134.6				
	PerProc		1194	1084	178.8	166.9	992.8	389.9	**170.4**	91.9
		PerBlock	509.3	470.9	110.8	100.5	601.2	232.6	113.8	56.2
		Stealing	**1254**	**1117**	155.3	**174.7**	943.8	403.1	168.8	**94.7**
		Donating	993.1	852.5	151.7	152.6	587.5	249.7	117.4	59.3
Local		Small	1114	993.5	153.7	149.2	593.2	297.4	121.0	68.8
		Medium	1220	1114	157.9	154.1	789.6	324.2	145.6	74.3
		Large	1219	1115	157.7	154.0	**1035**	362.0	155.3	82.8
		Huge	1221	1115	157.7	153.9	925.0	**418.8**	160.2	93.8

reduces for this example. The reason for this decline is the reduced ability to perform load balancing between blocks. In terms of optimal memory assignment for locally buffered queuing, the picture has also changed from the previous test. Assigning too much memory to local queues, can strongly reduce performance in this example. Too big local queues completely disable global load balancing, leading to starvation of entire blocks. As item data fills up the local queues quickly, the maximum amount of shared memory works well for these cases still.

The maximum performance achieved for the different tests and devices is shown in Figure 5.7. The highest absolute performance is achieved by the GTX TITAN in all tests, with up to 1250 GFLOPS and 185 GB/s. The GTX 680 achieved 1100 GFLOPS and 164 GB/s at its best. The GTX 580 achieved 859 GFLOPS and 125 GB/s. However, the best performance in comparison to the device's peak performance for FMA was achieved by the GTX 580 with 50% peak performance, followed by the GTX 680 with 34% and the GTX TITAN with 25%. We recorded the lowest FMA performance ratio fir tge GTX 680 in the items test with big items. In general, the big item test is clearly memory bounded. It seems that on older devices with compute capability 2.0 it is easier to get closer to the peak performance. The memory transfer rates were more uniform. For package data, all devices offered 50 to 70% of the memory bus bandwidth to the algorithm. For item data, the GTX 480 and GTX TITAN offered between 28 and 61%; the GTX 680 offered between 7 and 85%. Overall, it seems that the GTX 680 is the most unpredictable device and runs into memory bandwidth problems quickly. The high performance of the GTX 680 can only be achieved when using queues in shared memory and thus avoid accessing global memory as much as possible.

Table 5.4: Recursive expanding tests - performance (GFLOPS and GB/s) on a GTX 580

			Packages				Items			
			FMA		Memory		FMA		Mem	
			Small	Big	Small	Big	Small	Big	Small	Big
Linked	Mono		318.0	315.1	79.1	77.0				
		PerBlock	120.3	102.3	32.2	43.5				
		Stealing	546.5	397.8	119.8	96.5				
		Donating	266.0	182.5	67.8	46.3				
	PerProc		318.7	317.1	78.2	78.5	1.3	1.3	0.3	0.3
		PerBlock	159.5	170.1	47.0	49.1	6.6	5.5	1.7	1.4
		Stealing	541.6	427.2	119.5	100.7	6.9	5.4	1.7	1.3
		Donating	251.5	194.4	66.4	49.9	6.4	5.0	1.6	1.3
Block	Mono		37.5	60.7	11.9	15.9				
		PerBlock	165.8	144.6	48.0	55.6				
		Stealing	715.3	640.7	121.5	116.1				
		Donating	310.2	274.3	81.0	68.7				
	PerProc		81.2	74.8	20.1	18.7	2.4	3.7	0.6	0.9
		PerBlock	220.0	210.3	58.5	53.0	82.6	78.5	21.0	20.9
		Stealing	681.8	531.4	120.1	105.3	87.8	78.3	22.3	21.3
		Donating	287.1	271.2	74.3	67.2	92.6	76.5	24.0	19.8
CAS	Mono		148.9	150.2	36.7	36.2				
		PerBlock	166.4	144.0	46.6	54.1				
		Stealing	732.5	588.2	123.5	110.5				
		Donating	305.1	240.5	67.2	60.8				
	PerProc		130.8	133.8	31.2	32.0	4.3	8.2	1.1	2.0
		PerBlock	213.8	198.7	62.5	58.1	202.0	173.4	53.9	46.2
		Stealing	677.1	507.9	121.3	100.9	210.2	168.1	56.8	45.0
		Donating	280.3	265.9	74.9	69.0	191.0	174.3	50.4	46.6
Dist	Mono		785.1	**685.8**	**125.7**	**120.8**				
		PerBlock	155.1	131.0	42.6	52.8				
		Stealing	722.9	667.5	124.8	119.1				
		Donating	319.7	238.5	73.6	67.0				
	PerProc		716.1	635.6	124.9	120.1	223.7	180.5	52.9	46.1
		PerBlock	212.4	197.4	63.2	56.8	377.1	188.7	91.9	51.5
		Stealing	**841.4**	627.9	123.9	118.8	394.8	198.7	97.0	53.5
		Donating	274.3	272.6	76.1	69.1	377.3	184.3	95.2	50.5
Local		Small	828.8	629.6	124.9	115.4	192.6	145.3	49.6	36.5
		Medium	438.7	684.4	88.0	118.8	246.8	158.5	64.2	41.7
		Large	184.1	361.6	54.3	81.9	341.8	181.6	79.6	47.0
		Huge	184.0	167.9	54.4	47.0	**467.0**	**208.7**	**103.8**	**54.6**

Table 5.5: Recursive expanding tests - performance (GFLOPS and GB/s) on a GTX 680

			Packages				Items			
			FMA		Memory		FMA		Mem	
			Small	Big	Small	Big	Small	Big	Small	Big
Linked	Mono		252.0	270.5	56.8	61.0				
		PerBlock	80.1	69.4	18.1	17.3				
		Stealing	291.3	253.8	66.2	59.3				
		Donating	140.4	114.6	30.0	26.6				
	PerProc		254.6	259.7	56.7	59.0	1.7	1.6	0.4	0.4
		PerBlock	77.6	99.4	17.7	23.1	6.6	5.1	1.6	1.3
		Stealing	263.9	245.9	65.2	54.8	6.8	5.2	1.7	1.3
		Donating	133.5	111.7	32.3	26.4	6.6	5.1	1.6	1.3
Block	Mono		2.3	6.0	0.6	1.5				
		PerBlock	10.1	26.4	2.5	6.6				
		Stealing	43.1	69.1	10.2	16.9				
		Donating	13.2	35.7	3.3	8.7				
	PerProc		3.8	5.5	0.9	1.4	0.2	0.3	0.0	0.1
		PerBlock	26.6	38.4	6.5	9.1	2.0	3.8	0.5	0.9
		Stealing	74.5	70.8	17.9	17.0	2.9	3.2	0.8	0.8
		Donating	17.9	47.3	4.6	11.2	1.4	3.9	0.4	1.0
CAS	Mono		17.0	32.1	4.1	7.9				
		PerBlock	41.4	66.9	11.0	13.7				
		Stealing	132.8	164.1	32.0	38.6				
		Donating	54.4	75.5	12.0	18.7				
	PerProc		17.2	32.9	4.2	7.9	0.6	1.2	0.2	0.3
		PerBlock	56.1	72.5	10.8	16.3	7.6	13.0	1.9	3.2
		Stealing	158.5	155.0	39.6	36.9	12.8	14.3	3.2	3.5
		Donating	49.3	82.5	13.3	19.4	5.9	9.4	1.5	2.3
Dist	Mono		114.5	176.8	27.4	41.1				
		PerBlock	34.2	50.5	9.4	11.3				
		Stealing	107.0	164.2	26.1	38.6				
		Donating	53.3	79.2	13.5	19.5				
	PerProc		113.4	175.1	27.2	41.3	47.1	33.5	10.8	8.1
		PerBlock	44.1	71.5	8.4	16.5	48.4	32.7	10.1	7.9
		Stealing	129.9	166.1	25.8	39.1	44.0	33.7	10.1	8.3
		Donating	53.7	76.6	15.6	18.0	31.5	22.9	7.3	5.4
Local		Small	456.8	373.6	97.5	83.8	59.6	43.2	14.6	10.6
		Medium	**524.8**	**670.7**	**101.1**	**126.5**	69.1	46.2	16.5	11.2
		Large	232.6	446.8	54.6	91.0	108.3	47.8	25.4	11.8
		Huge	228.7	192.6	54.1	47.0	**148.7**	**59.0**	**38.8**	**13.7**

Table 5.6: Recursive expanding tests - performance (GFLOPS and GB/s) on a GTX TITAN

			Packages				Items			
			FMA		Memory		FMA		Mem	
			Small	Big	Small	Big	Small	Big	Small	Big
Linked	Mono		410.9	407.0	88.3	92.0				
		PerBlock	296.1	388.6	66.1	76.0				
		Stealing	1044	843.7	153.5	167.2				
		Donating	469.9	334.7	100.6	81.3				
	PerProc		417.5	409.3	102.3	101.0	1.7	1.7	0.4	0.4
		PerBlock	501.5	367.9	91.1	84.4	7.3	5.9	3.7	1.5
		Stealing	1157	884.5	149.1	166.6	12.6	9.7	3.2	2.5
		Donating	495.2	367.5	95.1	94.9	7.6	5.7	1.7	1.5
Block	Mono		78.3	74.0	18.0	16.1				
		PerBlock	224.4	274.0	43.8	59.7				
		Stealing	694.4	619.7	149.3	144.3				
		Donating	344.6	334.0	88.6	71.2				
	PerProc		93.6	83.6	20.9	19.8	5.1	5.4	1.3	1.3
		PerBlock	309.4	250.0	76.5	59.8	61.2	58.7	15.3	14.5
		Stealing	793.5	692.5	128.5	141.8	52.9	51.5	13.9	13.2
		Donating	361.2	368.3	81.7	83.2	62.0	57.1	15.1	14.6
CAS	Mono		329.3	340.3	71.7	74.5				
		PerBlock	252.2	303.9	58.2	63.2				
		Stealing	965.6	846.4	174.2	170.7				
		Donating	476.1	417.7	123.0	76.2				
	PerProc		337.2	342.2	82.4	82.2	17.2	24.7	4.3	6.1
		PerBlock	404.2	324.0	88.4	77.4	208.5	148.9	49.6	36.3
		Stealing	1074	961.6	144.6	168.8	230.4	187.8	55.7	45.5
		Donating	516.9	452.0	98.7	104.6	216.0	162.8	50.1	39.8
Dist	Mono		**1235**	1078	161.6	161.4				
		PerBlock	267.6	309.6	57.4	70.5				
		Stealing	1022	870.7	171.1	176.2				
		Donating	510.5	391.8	129.9	74.6				
	PerProc		1217	**1093**	**182.5**	170.4	748.5	334.8	142.3	79.7
		PerBlock	380.1	299.0	79.9	68.0	475.8	201.6	99.1	48.6
		Stealing	1223	1084	153.5	**177.1**	673.9	312.2	**151.5**	73.7
		Donating	538.9	428.7	106.0	99.0	463.8	210.8	97.5	49.9
Local		Small	1112	991.2	149.7	150.2	452.3	262.1	100.5	61.9
		Medium	494.5	1070	95.8	153.0	593.2	275.7	123.5	64.5
		Large	221.0	387.9	50.1	79.0	773.6	306.7	139.5	72.1
		Huge	221.3	169.4	51.0	40.0	**814.8**	**350.5**	147.6	**80.8**

Table 5.7: Performance comparison between the best results of the different devices for the standard recursive test and the expanding variant.

(a) Absolute performance in GFLOPS and GB/s

		Packages				Items			
		FMA		Memory		FMA		Mem	
		Small	Big	Small	Big	Small	Big	Small	Big
580	standard	854.4	702.6	125.2	120.3	569.9	280.8	117.9	74.1
	expanding	841.4	685.8	125.7	120.8	467.0	208.7	103.8	54.6
680	standard	1105.0	841.9	133.7	134.4	728.2	108.2	164.0	23.4
	expanding	524.8	670.7	101.1	126.5	148.7	59.0	38.8	13.7
TITAN	standard	1253.7	1117.0	185.3	174.7	1035.0	418.8	170.4	94.7
	expanding	1234.8	1092.6	182.5	177.1	814.8	350.5	151.5	80.8

(b) Performance compared to device peak performance

		Packages				Items			
		FMA		Memory		FMA		Mem	
		Small	Big	Small	Big	Small	Big	Small	Big
580	standard	0.50	0.41	0.65	0.63	0.33	0.16	0.61	0.39
	expanding	0.49	0.40	0.65	0.63	0.27	0.12	0.54	0.28
680	standard	0.34	0.26	0.70	0.70	0.22	0.03	0.85	0.12
	expanding	0.16	0.21	0.53	0.66	0.05	0.02	0.20	0.07
TITAN	standard	0.27	0.24	0.64	0.61	0.22	0.09	0.59	0.33
	expanding	0.26	0.23	0.63	0.61	0.17	0.07	0.53	0.28

5.2 Priority-based scheduling

The previously presented queuing strategies are concerned with efficiency and load balancing only. They hardly offer any influence on execution order. If multiple algorithms should be executed concurrently, or stay within a tight time frame, it can be beneficial to influence the execution order of individual elements. The results of a certain algorithm might be needed as soon as possible, or one wants to make sure the most important computations for a single algorithms are computed first. To achieve these goals and fulfill the Softshell processing model requirements, individual priorities for queue elements need to be supported.

5.2.1 Sorted Monolithic Queuing

On the GPU, the high overhead of synchronization primitives prohibits the use of complex data structures like sorted linked lists or heaps. Thus, we try to continue using a ring buffer-based queue and separate the insertion and removal from priority management. To follow the requested priorities, we us a separate controller block to periodically sort the queue according to the element priorities.

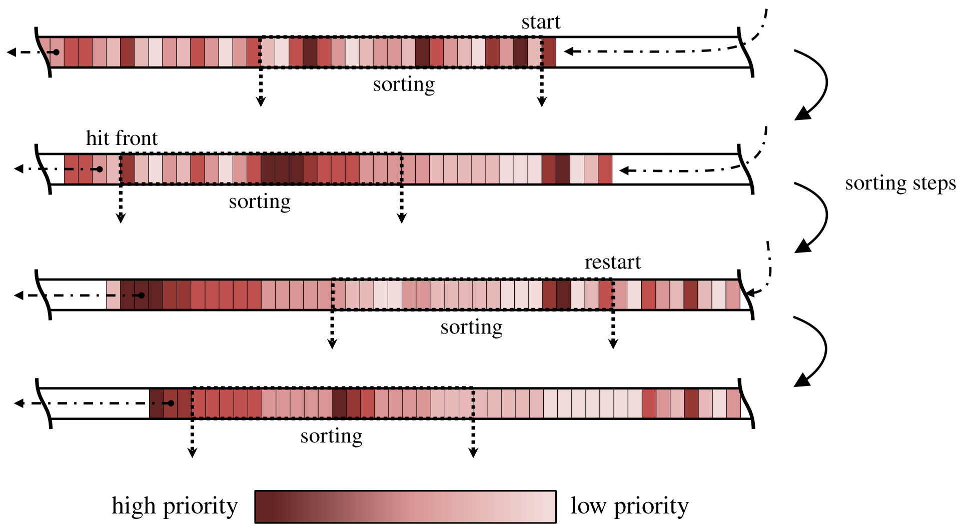


Figure 5.5: The priority-sorted monolithic queue supports insertion and removal of elements while the queue is iteratively sorted. In each iteration, a window is locked and sorted. For the next iteration, the window is moved half the window width to the front, taking high-priority elements along. If there is not enough time for sorting the front-most segment, the algorithm is restarted at the back.

Because the queue-sorting algorithm runs periodically, the main objective of the sorting is to move high-priority elements to the front of the queue. The order of elements at the back is not relevant as long as these elements are not removed from the queue before the sorting algorithm is restarted. To avoid occupying the execution of the controller block for too long, the sorting algorithm operates on a small window at a time. Sorting starts at the back of the queue, moving the most important elements to the front of this window. In the next step the sorting window moves half a window to front, taking the most important elements along. The window is moved to the front until the space between the window and the front of the queue has shrunk so far that sorting might not complete before the front of the queue hits the sorting window. If this situation arose, sorting would stall the execution of the entire system, as all elements used during sorting need to be locked. Locking elements during sorting can be achieved with a single pointer indicating the position of the current sorting window, or individually locking the individual elements if the Distributed Locks Queue is used. If the space between the window and the front of the queue becomes too small, sorting restarts at the back, moving newly inserted elements to the front. The algorithm is outlined in Figure 5.5.

To sort elements in parallel, we use bitonic sort [5], which is well-suited for a small number of elements. To perform the sort efficiently, each priority is loaded (or computed) by a single thread and stored along with the position of the element in the queue in shared memory. After locally sorting the data according to the priorities, the queue elements are rearranged within the queue according to the sorting result. Most often, the algorithm will deal with segments already sorted in previous iterations. In this cases, we can accelerate the computation, leaving out about one third of the bitonic sort network.

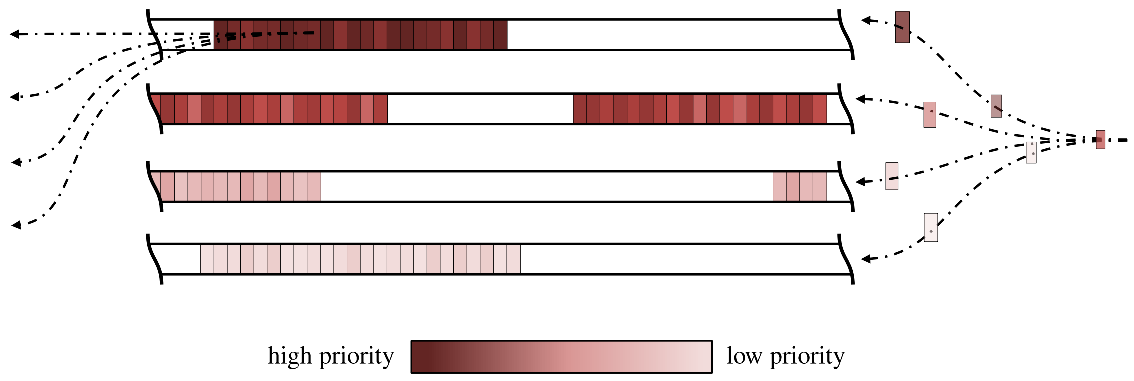


Figure 5.6: Priority bucket queuing provides a fixed number of buckets for individual priority ranges. Among engueue the priority of an element is evaluated and the element is inserted into the respective queue. Elements are taken out of high-priority queues first.

To perform parallel sorting on the queue, concurrent access to successive elements must be granted to the sorting algorithm. Additionally, enqueuing and dequeuing must not be disabled during sorting. Thus, only CAS ordered, distributed locked, and pointed queues can be used as underlying queues. While in case of CAS-ordered and Distributed Locks Queues entire queue elements must be moved during sorting, only pointers must be moved if a pointed queue is used.

The biggest advantage of constantly sorting the queue is the support for *dynamically changing priorities*. Of course, sorting requires *additional bandwidth and processing power* and thus might reduce the performance of the overall algorithm. If the queue grows large, sorting might *not be able to move high priority elements to the front of the queue fast enough* and fail in following the requested priorities. The major risk of this sorting strategy is that the window is moved too close to the front of the queue and *might stall the entire system*.

5.2.2 Priority-bucket Queuing

The biggest disadvantages of the sorted priority queue are its additional bandwidth and processing power requirement and the delay of moving new elements from the back to the front of the queue. An alternative which does not suffer from these problems is the priority bucket queue. For this queue, we discretize the range of supported priorities and provide a separate queue for each priority, as shown in Figure 5.6.

During enqueue, the most fitting queue is chosen to hold the new element. During dequeue, the queues are searched according to their priority and elements are drawn from the queue with higher priorities first. One advantage of this strategy is that *newly added high-priority elements are immediately available*. The major disadvantages are that *only a discrete subset of priorities* are supported and that *priorities need to be static*, disallowing highly dynamic scheduling strategies. While there is *no additional bandwidth or compute power needed* for sorting the queue, the number of queues is multiplied, thus, *increasing the memory requirement and the time needed for searching through the queues* during dequeue. Especially, small discretization steps are used, the time required for searching the queues can become rather large. If multiple threads are available during dequeue, each thread can check the fill-level of one queue, reducing the number of successive dequeue attempts. To efficiently implement this strategy, we use a modified version of the prefix sum algorithm to communicate the results of the initial probing of queue fill-rates.

5.2.3 Scheduling Policies

Building on priority sorted queuing and bucket priority queuing, different scheduling strategies can be developed. While priority sorted queuing in principle supports every possible scheduling strategy, sorting the queue introduces a delay which does not allow to react to priority changes immediately. In contrast, the bucket priority queue can immediately react on incoming high-priority elements. However, priorities must not change dynamically. Still, a wide range of scheduling strategies can be employed using bucket priority queuing.

Fair Scheduling In a system that supports the execution of different processes, events, or procedures, one common goal is to provide a fair scheduler, assigning an equal amount of time to each entity. Such a strategy is only possible, if the time used by individual procedures can be measured and recorded. In Softshell, capturing the execution statistics is an integral part of the system. When a procedure is started, we check the per-multiprocessor timer and record its value. After executing a procedure, we query the counter again and compute the spent cycles. The cycle count can then be used to update multiple measurements, including the aggregate execution time consumed by the associated event.

If **priority sorted monolithic queuing** is used, this aggregate can be put to use to dynamically update the priorities of all elements in the queue, to the inverse of the time frame used by event k:

$$p_k = \frac{\sum_i t_i}{t_k}, \tag{5.1}$$

where p_k is the priority, t_k is the time used by event k, and the sum runs over all events. In this way, events which only used little time in comparison to all other events receive a high priority and vice versa. If the priority-sorted queue was instantaneously sorted, all elements associated with the event that consumed the least time would always be in the front of the queue. As soon as the processing time of this event increased above another event, the other event was chosen for execution. However, as sorting lags behind, the scheduling also lags behind and those events whose elements are already in the front of the queue will temporarily receive more processor time. One interesting fact about this strategy is that even though the queue lags behind, the sorting strategy will group elements associated with the same event. Thus, elements of the same event will also be executed at the same time. While this will result in a burst-like change in the execution time distribution, it can have an overall positive effect, as elements of the same event might access similar memory locations and thus lead to a better cache usage. If this behavior is not desirable and elements of different events should be executed concurrently, a random jitter can be added to equation 5.1. In this setup, the scheduler will overall follow the previously defined priorities, but mix elements of various events.

Sometimes a fair scheduling strategy might not be desired and one rather wants to assign non-equal quotas to events. Such a policy can be achieved by slightly altering equation 5.1 to incorporate a target quota q_k for each event k:

$$p_k = \frac{\sum_i t_i}{t_k} \cdot q_k. \tag{5.2}$$

in this setup, p_k essentially corresponds to the ratio between target quota and the currently used quota. If an event overuses its quota, it will get a small priority and vice versa.

While this approach of assigning dynamically changing priorities to individual queue elements works with the priority sorted queue, **bucket priority queuing** would dramatically lag behind, as it evaluates the priorities only during enqueue. Still, a fair or quota-guided scheduling strategy can also be implemented using bucket priority queuing, as long as the number of different events is low. Then, it is possible to assign individual queues to individual events. According to equation 5.1 and 5.2, equal priorities are assigned to all elements associated with the same event. When assigning elements of each event to individual queues, the logical consequence is to dynamically change the priority of the queue and, in this way, of all elements in the queue. Hence, instead of relying on a fixed order of probing queues during dequeue, we can evaluate the priority of each queue before dequeue and draw elements from the most fitting queue.

As the evaluation of the queue priority involves accessing the accumulated time of all events, it is reasonable to store the priority for all queues in shared memory and only reevaluate the priorities in certain intervals. For this strategy, a jitter can also be used, adding the jitter to the priority of the individual queues. Even with the queue priorities stored in shared memory, a jitter can be added for each individual dequeue. If the procedure execution time is known to be equal for all events, or if a fair scheduling among the number of executed elements is desirable, the strategy can be even simplified. Instead of tracking the execution time of all events, we can rely on a uniformly distributed random number to choose the queue.

Shortest Job First Another common scheduling strategy on the CPU is shortest job first or its preemptive counterpart shortest remaining time. The goal of this strategy is to execute the job, or in case of Softshell, the event with the shorted execution time first. Again building on the meta data collected by Softshell, this scheduling policy can be implemented in a straight forward manner. The required meta data is the expected execution time $t_{exp,k}$ for every event k and the time t_k already spent executing it. Using the following equation, the priority for each event can be computed:

$$p_k = \frac{1}{\max(\epsilon, t_{exp,k} - t_k)}, \tag{5.3}$$

where ϵ is a small positive number, making sure that the scheduling policy can also handle execution times estimated too short.

The main risk of shortest job first scheduling is starvation. In case short running events are continuously created, long running events might never get any processing time. Such a strategy could be useful if long running algorithms, like, a segmentation algorithm, are paired with user interaction. In this setup, the current state of the segmentation would be visualized, and in case could be influenced.

Using **priority sorted queuing** p_k can again be evaluated continuously for all individual queue elements. Assuming an instantaneously sorted queue and ignoring executing elements, the order of events and their relative priority is only changed when a new event arises. Thus, it is sufficient to compute the priority for each event only upon the creating of a new event and rely on precomputed information during sorting. As sorting is not instantaneous and there are always elements being executed, the ordering of events may change at any point in time.

Using **bucket priority queuing**, a similar strategy as used for fair scheduling can be implemented. Again, every event is assigned its own queue. Whenever a new event occurs, the priorities for all queues are reevaluated and stored in global or shared memory. Until a new event is created, each worker draws elements from the highest priority queue only. If the execution of the highest priority event is completed, the queue is marked for reuse by the next incoming event, and elements from the queue with the next highest priority are executed.

Earliest Deadline First In hard real-time scenarios, one common scheduling strategy is earliest deadline first scheduling. Every task, or event, is associated with a deadline. If this deadline is not met, the result of the execution is useless and system failure is to be expected. The idea behind earliest deadline first scheduling is to focus all processing power on the event with the earliest deadline. Ignoring the overhead of scheduling, earliest deadline first is viable until the load reaches 100%.

To implement this strategy with **priority sorted queuing**, we associate a priority p_k with each event k in the following way:

$$p_k = \frac{1}{d_k - t}, \tag{5.4}$$

where d_k represents the deadline for event k as a fixed point in time and t is the current time. The closer an event comes to its deadline, the higher its priority gets, reaching infinity when the deadline is met. After the deadline the priority will be negative. Thus, sorting will move those elements with the closest deadline to the front. If a deadline is not met, elements will be moved to the back of the queue. Note that again, the relative order among events will not change unless a new event arises (or a deadline is missed). An alternative way to set up the priorities for earliest deadline first scheduling is:

$$p_k = -d_k, \tag{5.5}$$

which has the same result on the scheduling, with the difference that the priority does not change when a deadline is missed.

Using **bucket priority queuing**, we can again build on the idea of assigning an individual queue to each event and draw elements for the highest priority queue. However, if the number of different events grows large, keeping an individual queue for each event becomes infeasible. Accepting small deviation from the perfect execution order, we can again build on discretization and use equation 5.5. As priorities are directly associated with time, we discretize the time into fixed intervals and provide one queue for each interval. Events are associated with intervals according to their deadline as given by equation 5.5. If the deadline of different events falls in the same interval and thus into the same queue, the individual elements of these tasks get mixed. Still, those elements with a close deadline will be executed first, when drawing elements from those queues with the closest intervals. As seen in equation 5.5 the priorities are not bounded and thus an infinite number of queues would be needed. However, as the time progresses, the time interval eventually get hit. If there are still elements in such a queue associated with a hit interval, deadlines might be missed. If time moves past an interval, the queue is not needed any longer and can be reused for later intervals. In this way, we can build a ring buffer of queues, always drawing work from the queue with the closest intervals first. The number of required queues for this strategy depends on the smallest time difference between an event's deadline and its occurrence. If an event with a very distant deadline arises, it must still be provided with a queue. If this queue is still in use for an earlier deadline the entire system breaks. The chance of this failure to happen can be decreased by either increasing the number of queues or by enlarging the intervals. The two options form a trade-off between increased memory consumption and the ability to discriminate between events with close deadlines.

5.2.4 Evaluation: Arbitrary Priorities

For the priority sorted and priority bucket queue, it is interesting how well these queues can follow the priorities. To evaluate the queues ability to handle arbitrary priorities for individual elements, we assign uniformly distributed priorities to each element. We create an initial number I of packages and insert them into the queue. During execution, we capture the order in which elements have been executed, writing their id to an atomically operated array C. Each executed package enqueues another element of random priority and then executes a fixed number of FMA operations, simulating the workload of the procedure. In this way, there are always approximately I elements in the queue, which shall be denoted Q. We continue this dequeue-enqueue-execute process, until a fixed number N of elements have been executed. After all elements have been processed, we evaluate the execution order and compute the achieved scheduling accuracy. The scheduling accuracy S is defined as

$$S = \frac{1}{N} \cdot \sum_{n=0}^{N} \frac{1}{I-1} \cdot \sum_{j=0}^{I} s_{n,j},$$

$$s_{n,j} = \begin{cases} 1 & \text{if } P\left(C(n)\right) \geq P\left(Q_n(j)\right) \wedge C(n) \neq Q_n(j) \\ 0 & \text{otherwise} \end{cases},$$

where $s_{n,j}$ captures if one pair of elements (i and n), which were concurrently in the queue, have been chosen according to their priority. For uniformly distributed priorities, the lower bound for the scheduling accuracy is about 50%. If all elements are executed in the correct order, the scheduling accuracy is 100%. As multiple blocks concurrently draw elements from the queue, 100% accuracy is never reached in practice, if more than a single block is running.

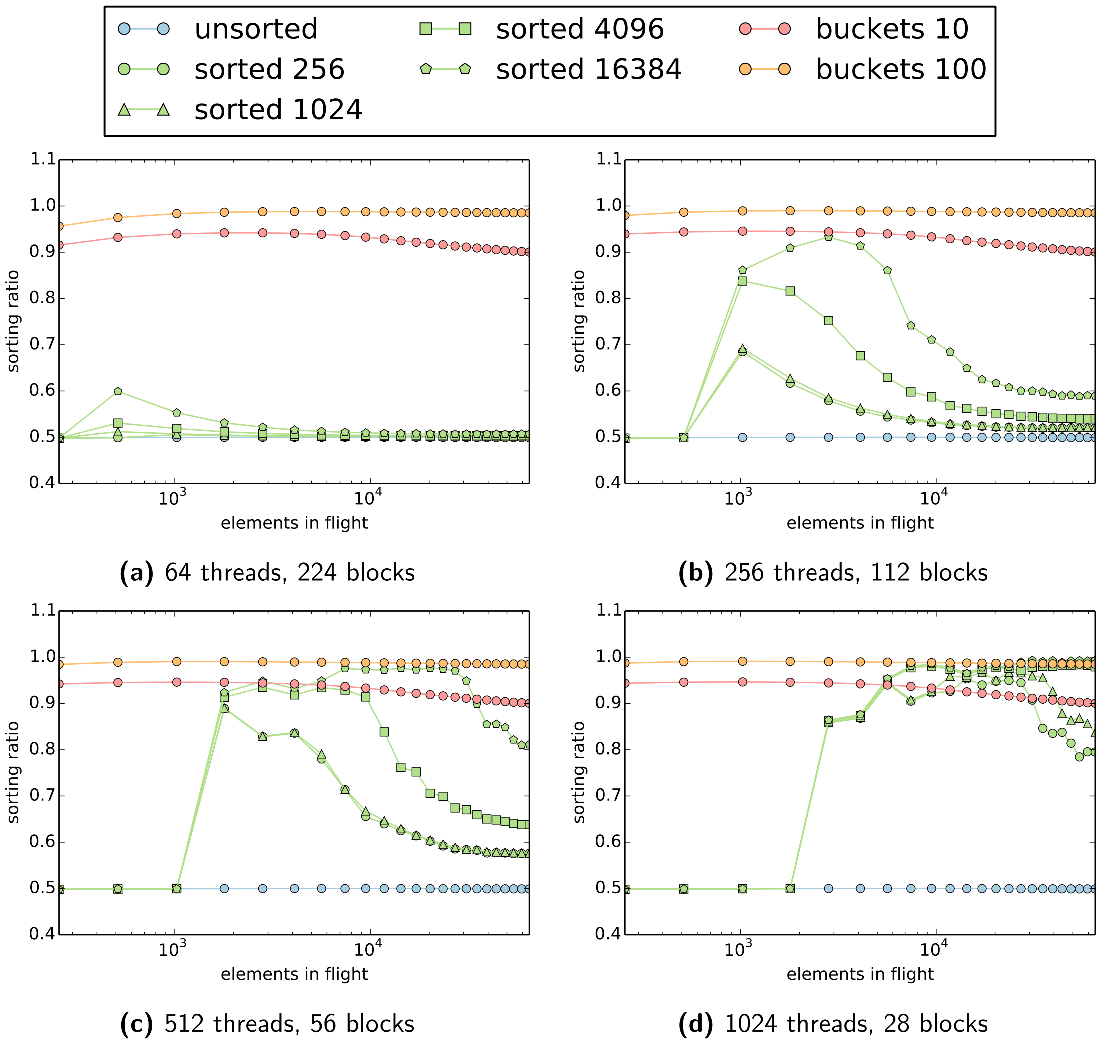
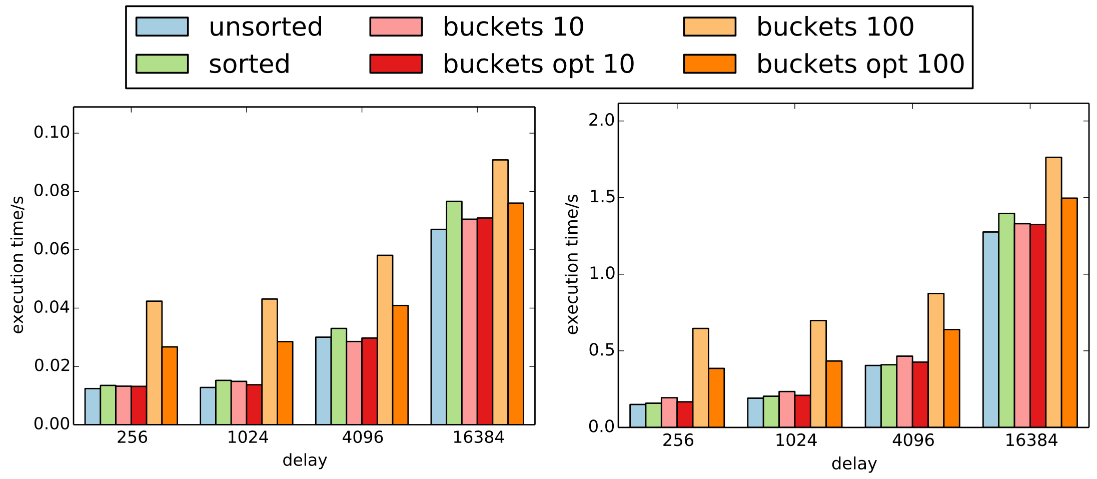


Figure 5.7: Scheduling accuracy on a GTX TITAN with varying procedure execution time (in number of FMA operations for the sorted queues) and number of threads per block. Bucket sorted queuing either uses ten or 100 buckets.

For the GTX TITAN, the scheduling accuracy achieved by the different queuing strategies is shown in Figure 5.7. Note that there was no interesting difference to be found between the GTX 580, GTX 680 and GTX TITAN, thus, we do not include plots for the GTX 580 and GTX 680. The scheduling accuracy plots clearly show the characteristics of the different priority scheduling strategies. If only a small number of threads are used for sorting, priority sorted queuing is not able to move high priority packages to front. Especially if there are many elements in the queue, the scheduling accuracy is equal to the unsorted queue. With increasing number of sorting threads, the sorted queue can adapt to the requested priorities. If the individual procedures have longer execution durations, there is more time for sorting and thus the accuracy also increases. If there are too many elements in the queue, high priority elements cannot be moved to the front of the queue fast



(a) GTX TITAN 64 threads

(b) GTX TITAN 1024 threads

Figure 5.8: Influence of priority scheduling on execution time.

enough and accuracy drops again. The accuracy of priority bucket queuing is independent of the execution time of the individual procedures and mostly independent of the number of elements in the queue. The scheduling accuracy slightly drops for an increasing number of elements in the queue due to the discretization of priorities. After executing many elements, the high priority buckets are nearly empty and elements are piling up in the low priority buckets. As the priorities are discretized, the queuing strategy cannot distinguish between the remaining elements well, and the accuracy drops. Overall, the error in scheduling accuracy of priority bucked queuing equals approximately the discretization error. If ten buckets are used, the scheduling accuracy is about 90%; if 100 buckets are used the accuracy increases to 99%. Priority sorted queuing only achieves such high accuracies if 1024 threads are used for sorting and about 10 000 elements are in the queue.

The impact of the different priority-based queuing strategies on execution time is shown in Figure 5.8. Priority sorted queuing adds a small overhead to the execution time compared to unsorted queuing. As the bucket queues do not execute a sorting algorithm, they can save bandwidth and execution power, which can be used for executing the algorithm itself. However, with an increasing number of buckets, the time spent on searching through the buckets has a significant influence on the execution time. Especially if 100 buckets are used, the execution time nearly doubles, if only few computation are executed by each procedure. The optimization for a concurrent queue lookup can reduce the overhead to some extent. With this optimization, the 10-bucket queue is the fastest priority queuing strategy, achieving a very good scheduling accuracy of 90%.

5.2.5 Evaluation: Quota-based Scheduling

To evaluate the ability of the priority queuing strategies to distribute execution time according to predefined time quotas, we implemented the scheduling policies as discussed in Section 5.2.3. We create four events with one procedure each and create 1000 initial

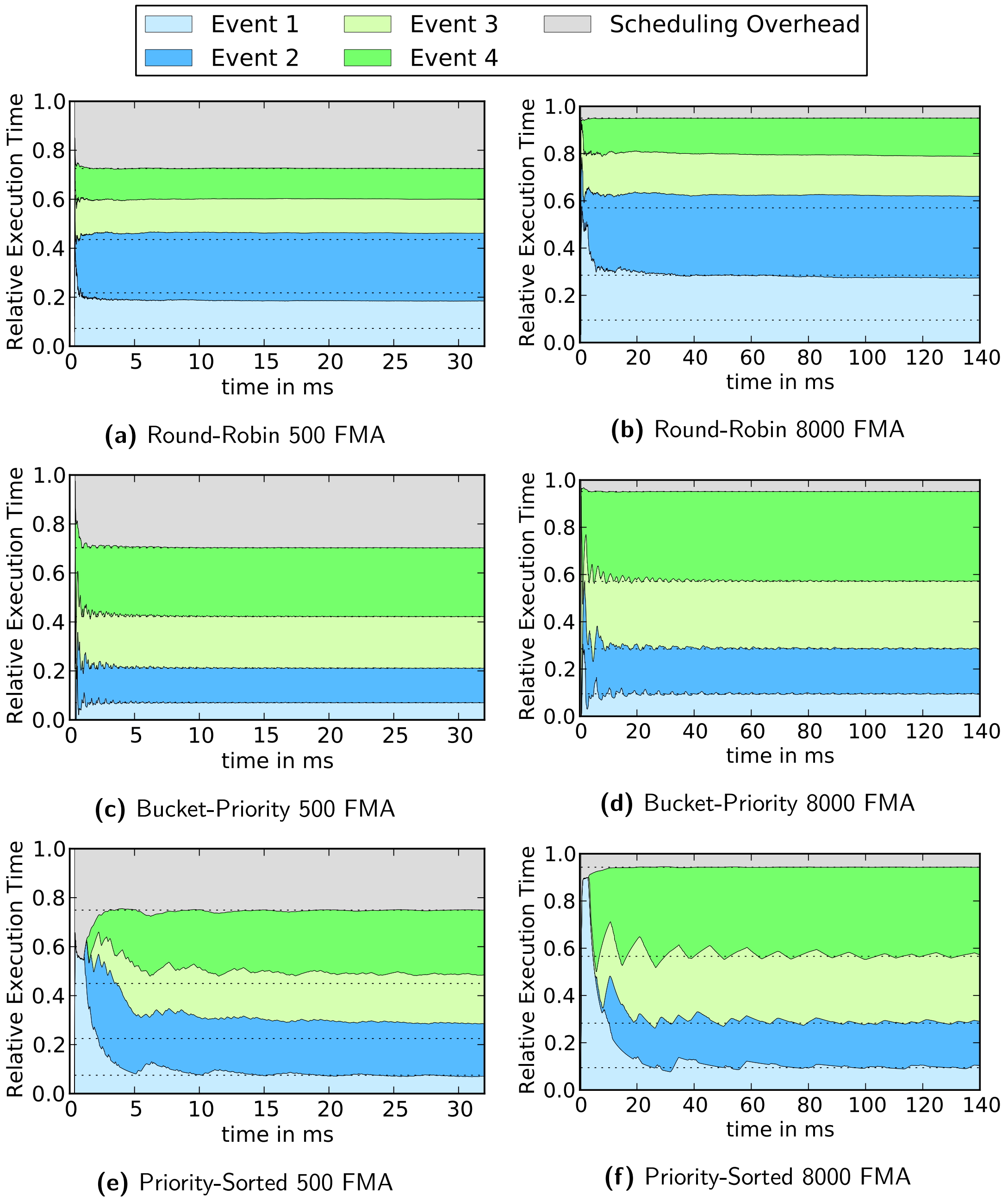


Figure 5.9: Quota driven scheduling with target time quotas of 10%, 20%, 30%, and 40% for event 1, 2, 3, and 4 respectively. The scheduling overhead does not increase when using smarter scheduling policies. While the buck-priority queue can quickly adjust the scheduling to the quota, the sorted queue takes significantly longer and oscillates around the targets. If the execution time of the individual procedures is too low (500 FMA), the sorted queue takes very long to get close to the requested quotas.

work packages for each procedure. Each procedure has different execution characteristics and after executing a randomly chosen number of FMA operations enqueues the same work package again. We capture the time spend on each executed procedure. Using these measurements, we estimate how well the scheduling policy is followed and how much overhead is used by the scheduler overall. The results of these tests are shown in Figure 5.9.

As a base-line for the scheduling overhead, we use a round-robin scheduler, which does not rely on any time measurements and simply selects procedures for one event after the other. The scheduling overhead corresponds to the entire time not spent on procedure execution, including searching through queues, dequeue operations and copying work packages for execution to shared memory. The scenario with an average of 500 FMA operations per procedure forms a challenging task. The scheduling overhead of the entire system is about 20%. With 8000 average FMA per procedure the scheduling overhead is about 2%.

If the quota driven scheduling either using bucket-priority queuing or priority sorted queuing, the overhead does not measurably increase. This fact indicates that the overhead is mostly due to dequeue operations and memory transfers.

The round robin scheduler does not know about the quotas and does not follow the demands (dotted lines). The bucket-priority queue is able to quickly adjust to the requested quotas, assigning exactly 10%, 20%, 30% and 40% of execution time to the individual events. After less than one millisecond the quotas are already matched well, if the procedure execution times are low (500 FMA). For 8000 FMA execution time, it takes approximately 7 milliseconds to converge. The priority-sorted queue takes significantly longer to converge to the requested quotas. With 500 FMA average load, the quotas are not fully met after 30 milliseconds, but close to the requests. The sorting algorithm can in this case not fully keep up, as there is not enough time to sort the queue. If procedures execute 8000 FMA on average, the scheduling converges in approximately 50 milliseconds. The burst-like execution behavior is clearly visible in this example, leading to an oscillation around the target quotas.

5.3 Discussion

Building on the combination of different queuing strategies and priority-based queuing, a variety of different scheduling strategies can be implemented. Distributed Locks Queues again proved to be the most efficient global queues. While per-block queuing (with stealing to achieve load balancing) can significantly increase performance for the slower queues, the Distributed Locks Queue only marginally profits from the reduced contention and thus hardly shows any performance gain on the most recent GPU architectures. Locally buffered queues can strongly boost performance, however, they can also hurt load balancing strategies and require well chosen queue lengths.

Per procedure queuing offers the ability to combine items and item sets for efficient execution and allows to react on suboptimal queue lengths. Using full individual priorities, even more advanced scheduling strategies can be implemented. With the proposed priority sorted and bucket-priority queue, it is possible to prioritize certain procedures or events, distribute processor time fairly, or establish real-time scheduling. Our synthetic tests indicate that even priority sorted queuing, which is able to handle dynamically changing priorities, operates with little overhead compared to a priority-free execution (**O4**). For a final validation of **O4**, we will show that dynamic priority scheduling also has a positive impact on real-world applications (Section 8).

Chapter 6

Execution of Diverse Procedures

Contents

6.1 Persistent Megakernels

One of the biggest challenges for a system following the Softshell specification is the ability to efficiently and autonomously execute different granularities of parallelism. Previous approaches of autonomous execution are all based on the *persistent threads* model [3]. Instead of using the available data parallelism and launching one thread per data element,

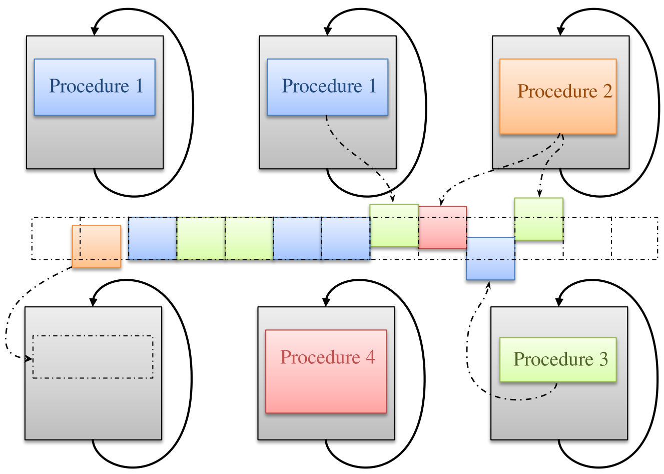


Figure 6.1: In the persistent megakernel model persistent worker blocks draw elements from a global work queue and execute this elements. New elements can be created and added to the queue. Different elements get mixed in the global queue and all procedures need to use the same number of threads for execution.

the persistent-threads model launches a single kernel, which exactly fills up the GPU. Each thread operates in an infinite loop as long as there is work available through a custom work queue. In every iteration of the loop, each thread draws an element from the work queue and executes it.

Concurrently, researches have proposed so called *megakernel* approaches. In a megakernel all program code is combined in one big kernel [105]. By combining persistent threads with a megakernel approach, it is possible to build an autonomous GPU system that is able to execute different procedures. For a straight forward combination, we simply insert a big switch statement into the body of the endless loop. Based on an identifier supplied by the queue element, the switch statements jumps to the procedure that should be executed. We call this approach *persistent megakernel*. It is outlined in Figure 6.1). In this way, all otherwise needed kernels can be combined into one *persistent megakernel*, which runs autonomously until the entire algorithm is finished. However, all previously proposed megakernel approaches are inflexible in terms of execution configurations and support one of the following granularities of parallelism.

The most straight forward megakernel approaches do not allow any cooperative strategies. Each element is expected to be executed by a single thread; the use of shared memory and synchronization barriers is strictly forbidden. In this setup, the megakernel block size can be chosen arbitrarily, and does not have any influence on the correctness of the execution. In a naïve approach, every thread starts by drawing an element from the queue, executes it and then draws the next element – until the queue is empty. As threads within the same warp could possibly draw different elements, this strategy leads to severe thread divergence [67].

Another common approach is to build on the fact that warps execute in SIMD lock step and in this way can cooperatively work on tasks without explicit synchronization. Megakernels based on this assumption normally force every queue element to be executed by exactly the number of threads that make up a warp. Shared memory can be used, if each warped is assigned its own region of shared memory. On current NVIDIA devices, warp voting and shuffle operations can be used to communicate. In a straight forward implementation, every warp operates independently, drawing a new element from the queue in every loop iteration.

The third way of setting up a megakernel is to have one entire block working on a single queue element cooperatively. In this way, shared memory and synchronization barriers can be used. Additionally, memory access patterns and cache usage can be optimized. In a simple implementation, one or possibly multiple threads are used to draw an element from the queue in the beginning of each iteration. The element is then stored in shared memory, followed by an explicit synchronization, so that all threads within the block have access to the element. In this approach all procedures must use the same number of threads.

In terms of performance, the major problem of persistent megakernel approaches is different procedure characteristics. If a lightweight procedure is combined with a resource-hungry procedure the resource-hungry procedure determines the characteristics of the megakernel. For example, if one procedure requires four times as many registers as the other, the overall number of concurrently running blocks is four times as low, as if only the lightweight procedure was started. This reduces the ability of the GPU to hide memory

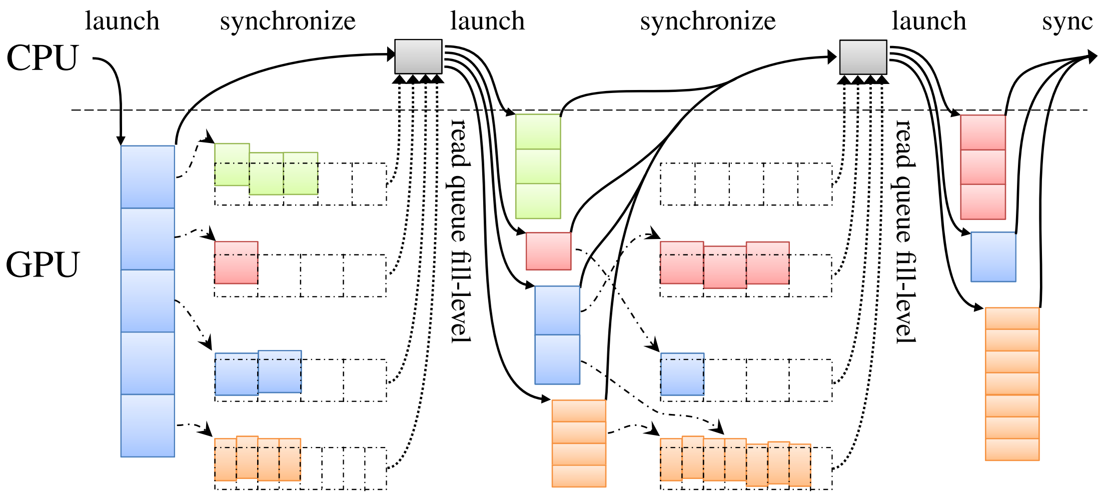


Figure 6.2: Kernels with queues are a simple extension to the kernel-based programming model, offering the ability to collect items and item sets for equal procedures and issue them for efficient execution.

latency. Thus, persistent threads approaches are preferable used for recursive algorithms, with only one type of procedure.

In the following, we propose three different strategies to build GPU systems supporting the execution of different granularities of parallelism, *i.e.*, work items, x-times, and work packages.

6.2 Kernels with Queues

The probably most straight forward approach for building a scheduling framework supporting all previously mentioned granularities using the traditional kernel-based programming model is the approached outlined in Figure 6.2. Every procedure has its own queue. To start the execution of procedures, the CPU reads back the queue fill-levels and issues individual kernels for all procedures. For work packages, a new block is launched for each queue element. For items, on thread is launches for each element in the queue. The block size is chosen to maximize occupancy. The same way, multiple item sets are grouped to be executed on the same block.

During the execution of procedures, new elements can be enqueued. After all kernels have finished their execution, the current queue fill-rates are read back from the GPU and new kernels are launched for each procedure. If all queue-fill rates are zero, the algorithm is finished. Note that this approach is similar, although more general, than the approach taken by Laine *et al.* [67] for path tracing.

By using multiple queues and individual kernels for each procedure not only different granularities of parallelism can be supported, the individual kernels are also optimized for the individual procedures and divergence is minimized. To achieve a good GPU utilization, even if the queue fill-rates are low, all kernels can be issued into different streams and thus be executed concurrently. As producers and consumer are within different kernel launches, the highly efficient collector queue can be used in this approach.

6.3 Hybrid Dynamic Parallelism with Controller

Dynamic parallelism is a big step for GPU-based algorithms as it makes GPU execution independent of the CPU. However, as mentioned previously, dynamic parallelism by itself does not work well with work generated for individual threads or warps. The granularity of generated work must at least be a block. Thus, we take a similar approach as with *kernels with queues*, combining dynamic parallelism with individual queues for each procedure and issue them for parallel execution. For issuing the collected elements, we use a separate controller kernel, which continuously polls the queue state and as soon as there are enough elements for filling up at least one block, launches a new kernel via dynamic parallelism. This approach is outlined in Figure 6.3.

For the controller we use a single warp. It launches new kernels into different streams to allow for parallel execution. Using a controller is essential: The alternative would be checking the queue fill-rate during insertion and as soon as sufficient items or item sets are in the queue launching a new block. However, if the number of queue elements never reaches this threshold, the algorithm would terminate without executing all elements. Consequently, we also need a method to stop the controller, when all elements have been processed. To achieve this goal, we use a counter of launches being in-flight, but not executed yet. The counter is increased by the controller and reduced by each worker block after completion. If there are no more elements in the queue and the in-flight counter is zero, the controller can also terminate. The in-flight counter can also be used to decide whether non-full blocks of items or item sets should be launched. If there are many launches in flight, there is no reason to start partially empty workers. If there are hardly any workers executing, it is reasonable to start inefficient workers to produce more work and fill up the device.

As there is no synchronization between adding elements to queues and launching kernels, the queues need to be safe for concurrent insertion and removal. To achieve high performance we use a distributed locked queue for each procedure, which especially on most modern devices is very efficient when accessed by thousands of threads concurrently. Note, that there are multiple delays involved with launching a new kernel with dynamic parallelism, thus a simple collector queue as used for kernels with queues often appears to be safe for execution. This is not the case, as kernels can get launched for elements which have not been written to the queue yet.

The advantage of this approach in comparison to kernels with queues is the independence of the CPU and the ability to start procedures while others are still executing. The advantage over megakernels is that the kernels are adjusted for each procedure, leading to perfect register count and shared memory usage, which might lead to a better utilization. The downside is that fast on-chip shared queues cannot be used, as data cannot be transmitted via shared memory to new kernel launches.

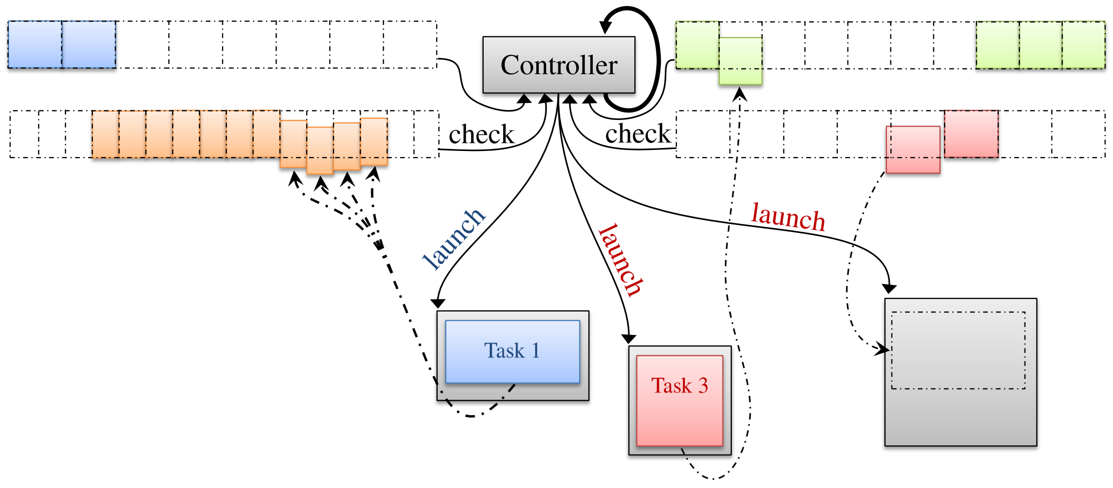


Figure 6.3: Hybrid dynamic parallelism with controller is an adaption of kernels with queues for dynamic parallelism. Instead of controlling the kernels launches from the CPU, a controller block on the GPU continuously scans through the queues and launches kernels to execute the newly added elements.

6.4 Versatile Megakernel

To support items, item sets and workpackages, we propose a versatile megakernel approach which dynamically assigns the available threads to the incoming elements. Again, we use a queue for each procedure to avoid thread divergence, as shown in Figure 6.4. To have a sufficient number of threads available for execution, we synchronize all threads in the beginning of each persistent threads loop iteration. Cooperatively, they then try to draw equal procedures from the queues. For items, we draw as many procedures as there are threads in the megakernel block. For item sets, we first compute how many elements can be execute in each warp and then try to fill up all available warps. For packages, the easiest approach would be to draw a single procedure and use the maximum number of threads required among all packages. This strategy is inefficient, if the number of threads needed for different procedures varies strongly. As the package with the largest number of threads t_{max} determines the block size of the megakernel, the utilization of the megakernel would reduce to

$$U_{min} = \frac{t_{max} - t_{min}}{t_{max}},$$

during the execution of the package procedure with the lowest number of threads t_{min}.

To increase the minimum utilization, we execute multiple package procedures concurrently in the same megakernel block. Similar to the execution of multiple item sets, we draw as many package procedures from the queue as can be executed concurrently with the available number of threads. As individual subsets of threads execute the procedure, the default synchronization function which synchronizes an entire block cannot be used. However, in CUDA PTX language, a variant of the synchronization primitive is available. The PTX synchronization can use one of 16 barriers to synchronize an arbitrary number of

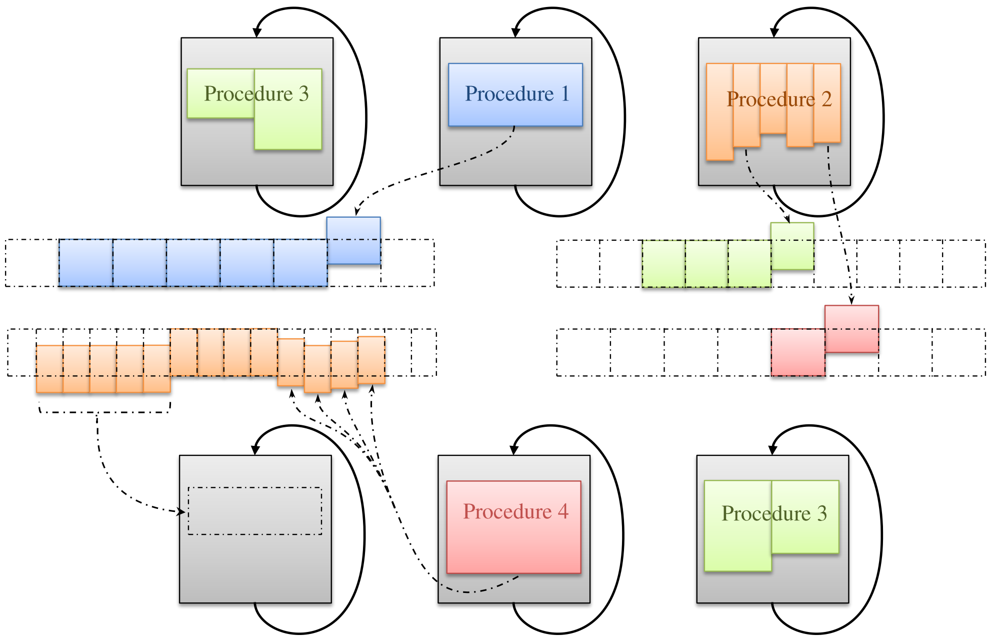


Figure 6.4: Our versatile megakernel dynamically adjusts its thread setup to the incoming elements. To fill up the worker blocks it also executes multiple packages concurrently while providing access to individual shared memory regions and supporting individual synchronization.

warps. Within package procedures, we use the barriers 1-15 and assign them to individual packages, as outlined in Figure 6.5. Based on the known number of threads used for package execution, the right barrier can be computed efficiently. In the megakernel itself, we used barrier 0 to synchronize all threads within the block.

To further increase the utilization achievable by the versatile megakernel, we choose the block size as the smallest common multiple of all package thread counts. Due to the block size limit, the smallest common multiple might be too high, and it might not be possible to achieve full utilization for all procedures. We then evaluate all possible block sizes and choose the one yielding the highest minimum utilization. In this way, we are able to build a versatile megakernel which is able to execute input data for individual threads, small groups of threads, and blocks of arbitrary size, while supporting synchronization within these blocks and maximizing utilization.

In addition to the required threads, procedures might also require the use of shared memory. To avoid excessive shared memory requirements, we dynamically assign shared memory to the executed procedures. Besides the number of required threads, we expect each procedure to provide its required amount of shared memory during compile time. When deciding on the block size of the megakernel, we also consider the required shared memory for the optimal block size and launch the megakernel with sufficient memory. In each megakernel loop iteration, before calling the procedure, we assign individual regions of shared memory to each package. During the next iteration, the same shared memory region can be used for the execution of a different procedure.

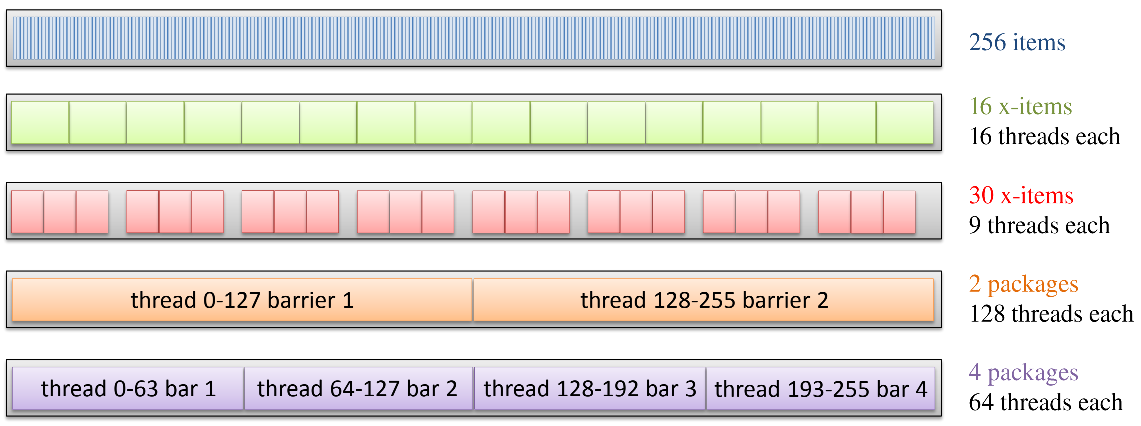


Figure 6.5: Different granularities of parallelism can are executed by our versatile megakernel. Individual threads are used for items, item sets are grouped to fall within warp boundaries and individual packages are supplied with individual synchronization barriers.

Another important application for shared memory is as storage for input elements. Especially, when multiple threads work on the same input element, *i.e.*, for work packages and item sets, keeping a single copy of the element in shared memory might be preferable to reduce the number of needed registers. If shared memory is also used by the procedure itself, the decision of whether to store input elements in shared memory or in registers becomes a tradeoff. According to our experience, especially for larger elements, using shared memory is preferable to registers. Another application for shared memory in persistent thread approaches is queues. As shared memory is magnitudes faster than global memory, local queuing strategies in shared memory are attractive for persistent megakernel approaches, which even further increases the pressure on shared memory. To reduce the shared memory requirements, queues that offer access to the elements in the queue for longer periods can be used and thus both, shared memory and registers can be saved.

6.5 Evaluation

To compare the performance of the different megakernel approaches, the dynamic parallelism variants and kernels with queues, we tested all presented methods with various optimization strategies in three different use case scenarios. The first is a simple recursive algorithm which re-issues the same procedure with declining probability. The second algorithm is a tree traversal algorithm, with different types of procedures, simulating an expanding algorithm. The third is a pipeline simulation with strongly varying characteristics among all stages, including execution time, register counts, thread count, and the use of items, packages and item sets.

The baseline for all tests is given by the kernel with queues approach (KwQ). The KwQ baseline implementation at first copies the input elements from the queue to shared memory in a cooperative fashion and then executes them. KwQ_A does not take the additional effort of copying data to shared memory, it leaves the elements in the queue and lets the procedures directly access them. As a final optimization, we launch the kernels in different streams (KwQ_{A+S}).

For dynamic parallelism we provide similar optimizations. The most straight forward dynamic parallelism implementation launches a new kernel whenever a new procedure becomes available (DP). As this strategy can only handle packages efficiently, we omit results for itemized tests, which most often did not deliver any results, probably due to the shear number of kernel launches. HDP stands for our hybrid dynamic parallelism implementation as outlined in Section 6.3. It uses a distributed locked queue to collect elements and copies the elements to shared memory before execution. Similar to KwQ_A, HDP_A provides the executing procedures with direct access to the elements still in the queue. Both hybrid dynamic parallelism approaches use streams to allow for current execution.

For the megakernel approaches, we distinguish between three major variants: the basic persistent megakernel (PMK) supports only one granularity of parallelism at a time and does not merge equal procedures. Our basic versatile megakernel VMK_B executes multiple items and item sets concurrently, but only one package at a time; the full versatile megakernel (VMK), as presented in Section 6.4 additionally executes multiple packages concurrently to increase utilization. One big advantage of persistent megakernel approaches is the use of queues in on-chip shared memory, which we also evaluate and indicate with a $_{shared}$ suffix. Again, we also switched between copying queue elements to shared memory and accessing them directly (VMK_A).

6.5.1 Overview

To provide a general overview of all investigated and mentioned scheduling approaches, we list their most essential characteristics in Table 6.1. While the kernel-based execution model (K), and dynamic parallelism (DP) are well optimized for the execution of complete kernels, they lack the ability to collect work for individual threads. With our adjustments, we provide this missing feature and support the efficient execution of items and item sets in the kernels with queues (KwQ) and hybrid dynamic parallelism (HDP) implementation. However, these approaches do not allow inter-block synchronization or fast on-chip queuing. DP and HDP run autonomously on the GPU, but introduce the overhead of book keeping. K and KwQ require the synchronization with the CPU for kernel launches, but do not require any book keeping. The basic megakernel approach MK as, *e.g.*, used in Optix [105], only offer a small feature set and suffer many performance problems (divergence due to the disability of combining equal procedures, no fast-on chip queuing, no individual optimization). Persistent megakernels (PMK) are a combination of a simple megakernel with persistent threads, which show a richer feature set, but only efficiently execute one granularity of parallelism during one launch. Our versatile megakernel (VMK) has a full feature set. It's only shortcoming is the fact that the kernel launch is not optimized for individual procedures.

One previously unmentioned ability is the influence on scheduling. While most approaches greedily execute the available work, PMK and VMK have the ability to adjust their scheduling policy to the current need. As these two approaches only remove elements from the queue directly before execution, they are free to reorder the queue elements or choose the most fitting for execution. Especially static scheduling policies are easily set up for PMK and VMK: Prioritization of a certain procedure over another can be achieved by drawing elements from queues of high priority procedures before searching other queues for elements.

Table 6.1: Characteristics of the execution models used for procedure scheduling on the GPU: Simple host-controlled kernels (K), host-controlled kernels in combination with a queuing approach (KwQ), dynamic parallelism (DP), our hybrid dynamic parallelism (HDP), megakernels (MK), persistent megakernels (PMK), and our versatile megakernel (VMK).

	K	DP	MK	PMK	KwQ	HDP	VMK
autonomous execution		●		●		●	●
schedule item			●	●	●	●	●
schedule x-items					●	●	●
combine equal elements					●	●	●
schedule packages	●	●		●	●	●	●
inhomogeneous package size	●	●			●	●	●
thread synchronization	●	●			●	●	●
inter-block synchronization				●			●
individual optimization	●	●			●	●	
avoid book keeping	●		●	●	●		●
avoid read-back to CPU		●		●		●	●
adjustable scheduling							●
fast on-chip queuing				●			●

6.5.2 Recursive Algorithm

We use a simple single procedure recursive algorithm to start the evaluation. As recursive algorithms are a good fit for persistent thread models, we would expect the persistent approaches to perform well in this scenario. We provide two variants for this test: one operating on items, the other on packages. In both cases, we start a fixed number of initial packages / items and with a certain probability re-emitted them. For the first 20 iterations we use a re-emit probability of 100%; for every following iteration, we reduce the probability by 5% leading to a maximum iteration count of 40. To simulate the time required for the execution of individual items / packages, we either perform 512 fused multiply-adds (FMADs) to simulate arithmetic load or 64 memory transactions to simulate a memory bounded algorithm. Note that FMADs provide a high degree of instruction level parallelism. In addition, we also vary the size of work items (16 / 64 bytes) and packages (16 / 256 bytes).

The results of these tests are shown in Table 6.2. While simple kernel launches (K) perform reasonable well for packages, items require queues to execute multiple elements per block (KwQ) and achieve good performance. With this strategy we could boost performance by a factor of 10 to 20.

Table 6.2: Recursive test results for arithmetic (FMA) and memory accesses synthetic loads. Performance provided in GFLOPS and GB/s; Best performance marked bold.

(a) GTX 580

	Packages				Items			
	FMA		Memory		FMA		Mem	
	Small	Big	Small	Big	Small	Big	Small	Big
K	879.9	498.3	117.5	111.1	34.6	34.3	8.5	8.5
KwQ	874.3	496.2	117.3	111.4	345.0	329.1	78.5	78.7
KwQ_A	878.1	690.8	117.9	114.7	342.9	**343.3**	79.6	81.3
KwQ_{A+S}	877.6	691.1	116.1	114.2	343.7	339.6	79.4	81.2
PMK	896.6	672.3	124.9	**125.2**	463.8	228.8	88.0	62.1
VMK	898.2	676.1	124.8	124.7	404.0	226.0	88.5	61.0
VMK_A	884.5	670.1	**126.4**	122.3	419.7	300.4	99.8	**82.5**
VMK_{shared}	**987.3**	**763.0**	126.4	122.8	**559.6**	286.3	**114.6**	82.4

(b) GTX 680

	Packages				Items			
	FMA		Memory		FMA		Mem	
	Small	Big	Small	Big	Small	Big	Small	Big
K	256.1	207.8	58.0	49.2	5.1	5.1	6.3	5.4
KwQ	249.2	206.3	57.6	49.9	80.4	**88.8**	73.8	**110.8**
KwQ_A	268.6	282.4	60.0	62.1	82.3	68.7	82.3	87.5
KwQ_{A+S}	268.8	282.0	60.0	62.2	82.2	68.5	82.0	87.6
PMK	112.8	182.4	27.1	45.9	48.0	46.7	10.7	10.6
VMK	112.0	185.3	27.3	45.9	43.0	39.7	48.8	57.7
VMK_A	270.5	284.3	71.6	69.4	103.4	80.4	120.6	103.5
VMK_{shared}	**1172**	**940.2**	**154.0**	**141.6**	**147.9**	79.3	**131.6**	99.8

(c) GTX TITAN

	Packages				Items			
	FMA		Memory		FMA		Mem	
	Small	Big	Small	Big	Small	Big	Small	Big
K	955.7	483.3	142.1	110.1	31.7	27.7	6.6	6.1
KwQ	944.2	482.4	141.3	110.6	859.6	309.6	118.8	75.8
KwQ_A	919.8	836.2	139.9	133.5	874.1	382.2	120.9	83.0
KwQ_{A+S}	915.8	827.0	139.0	131.8	863.3	379.3	120.2	83.0
DP	28.5	28.5	7.5	7.6				
HDP	881.0	32.7	123.6	9.2	636.8	176.7	106.7	47.2
HDP_A	841.6	309.9	125.0	70.6	593.3	59.9	103.5	16.7
PMK	1098	951.7	**164.2**	155.1	869.0	369.0	149.2	90.9
VMK	1102	951.3	158.9	150.4	**1005**	389.9	147.9	92.8
VMK_A	1071	862.2	145.4	141.7	881.8	**437.8**	143.7	**98.4**
VMK_{shared}	**1145**	**1139**	161.1	**157.7**	934.0	339.5	**150.5**	79.4

As only equal items or equal packages are used in these tests, there was no difference between VMK_B and VMK and we thus only present the results for VMK. Due to the same reasons, PMK also achieved a very similar performance, with the exception of the item scenario with memory load on the GTX 680, where PMK clearly lagged behind. The difference between PMK and VMK for this simple scenario is that PMK stores a procedure identifier for each element in the queue, to distinguish between different procedures. VMK has one queue for each procedure and thus does not need an additional identifier. On the GTX 680 this small difference made a big difference if the memory bus was under a lot of pressure by the synthetic memory load.

On the GTX 580 the QwK_A approach achieved the best performance for the big item scenario with FMADs load. In all other tests, the megakernel approaches achieved higher performances than the kernel based approaches, leading to speedups of up to 44%. VMK achieved the best performance in all but one remaining scenarios. In this scenario, VMK was only 0.4% behind PMK. Among the VMK approaches, keeping queues in on-chip shared memory either achieved the best performance or was only slightly behind the best other megakernel approach. There was overall a small performance boost if big data was not copied to shared memory, but rather executed with direct pointers to global memory. If the data had been used more than once during procedure execution, the situation might have been reversed.

On the GTX 680 we see a different picture. KwQ again achieved very good performance for big items. In all other cases the shared memory queues within a VMK clearly outperformed all other approaches. For the package FMAD cases VMK_{shared} was about five times faster than all other approaches. For the remaining cases VMK_{shared} was two to three times faster than the best kernel based approaches and reasonably faster than the other megakernel approaches.

On the GTX TITAN, again, the situation is different. All approaches are closer in terms of performance. Neither kernel-based approaches nor dynamic parallelism based approaches achieved the best performance in any tested scenario. DP is much slower than its host controlled counterpart K (up to 40 times) and did not finish for items at all. With the HDP we could reduce the performance gap to a factor of $1.1 - 5$ times. However, KwQ always outperformed HDP In all cases, the megakernel approaches achieved the best performance. For package data, storing it in shared memory queues performed best. Small item data is handled efficiently, if it is either copied to shared memory before execution or directly queued in shared memory. For bigger data, directly accessing the data from global memory during procedure execution performed best, as it reduces the pressure on shared memory and in case increases occupancy. The overall gain of the megakernel approaches over kernel based approaches was between 14% and 36% and about 20% on average.

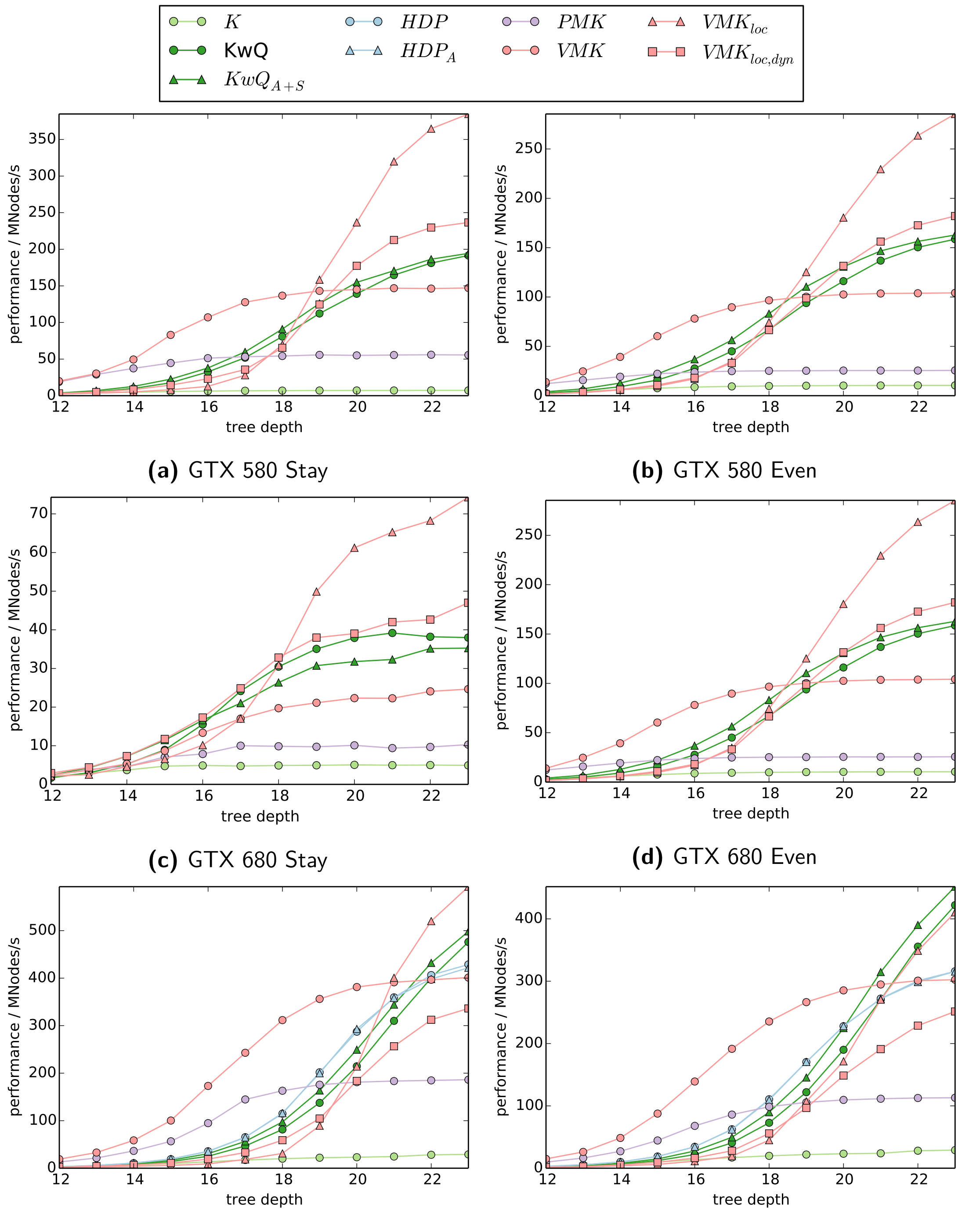
In terms of maximum performance, the GTX 580 achieved $0.9 TFLOPS$ / $126 GB/s$, the GTX 680 achieved $1.2 TFLOPS$ / $154 GB/s$, an the GTX TITAN achieved $1.1 TFLOPS$ / $161 GB/s$, which is 62% / 65%, 72% / 65%, and 47%, 56% of peak performance respectively. On the GTX 580 and 680 this high value is only possible due to queues in shared memory. On the GTX TITAN the queues in shared memory do not have such a severe impact on performance. However, all megakernel based approaches significantly improved performance over the other techniques.

6.5.3 Tree

The tree example is similar to the recursive use case, with the differences, that the number of generated elements are between 0 and 2, and that four different node types (which need to be executed with different procedures) are present in the tree. In the simplest version of the tree example (a fully filled binary tree) the algorithm starts with a single item, the root node. During the execution of every node, work items for two new nodes are generated until the maximum depth of the tree is reached. We model the probability of encountering a certain node based on the type of its parent. In the *Even* setup, every node type has the same probability, independent of the parent. In the *Stay* setup, the probability of encountering children which are of a different type is low (4%) and thus tree branches will enqueue items with the same queues all the time. In the *Change* setup, the probability of encountering a child of the same type is low (4%), which leads to alternating elements and non homogeneous enqueue behavior. We set the synthetic load for each node to 120 FMADs, which are computed using 24 registers. In the fully filled tree scenario, we expand the tree from depth 0 to depth 24 with 16.7 million tree nodes. Besides the fully filled tree, we also test a sparse forest. We start the algorithm with 65536 evenly distributed root nodes, each generating an individual tree. We choose the probability of encountering children in such a way that the number of nodes overall remains approximately constant.

Figure 6.6 shows the average number of executed tree nodes per second for a traversal from a single root node to a certain tree depth for a full filled tree. The *stay* scenario should be the easier scenario for the GPU, as threads are likely to enqueue the same node types, while in the *even* scenario the probability for each node is equal. As expected, on the GTX 580 and TITAN the *stay* achieved a higher performance than *even*. However, on the GTX 680 the situation is reversed and the overall performance for this test case is far below the GTX 580 and TITAN. On all devices and in all scenarios, the VMK variants were either achieving the highest performances or were only marginally behind the best technique.

There are a few general trends to be found in all tests: First, K with no method of collecting items cannot achieve good performance. Second, KwQ in all variants performed equally well. KwQ is having a harder time for lower tree depths, but when the tree gets large there is lots of parallelism available and KwQ can at some point outperform VMK with global queuing. When the tree is large, the read-back and kernel launch overhead is getting very small in comparison to the kernel execution time and thus the kernel based model is a better fit. Third, HDP slightly outperforms KwQ for lower levels; for larger trees KwQ slowly outperforms HDP. Forth, PMK performs modestly. As the four different node types introduce divergence, it most often is four times slower than its VMK counterpart, which avoids divergence completely. In the *stay* scenario PMK works a bit better as there is less divergence, as all created child nodes are equal. Fifth, VMK with global queues show a very good performance for low tree depth, as load balancing between worker blocks happens even when there are only few procedures in the system. For higher tree depths, queues in shared memory can significantly boost performance (especially on the GTX 580 and 680) and outperform techniques that go through global memory by a factor of up to two.



(a) GTX 580 Stay

(b) GTX 580 Even

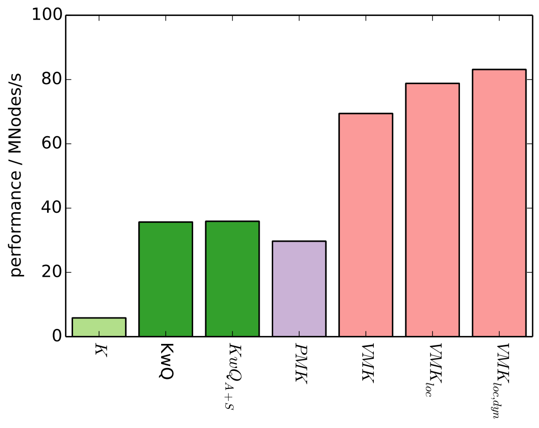
(c) GTX 680 Stay

(d) GTX 680 Even

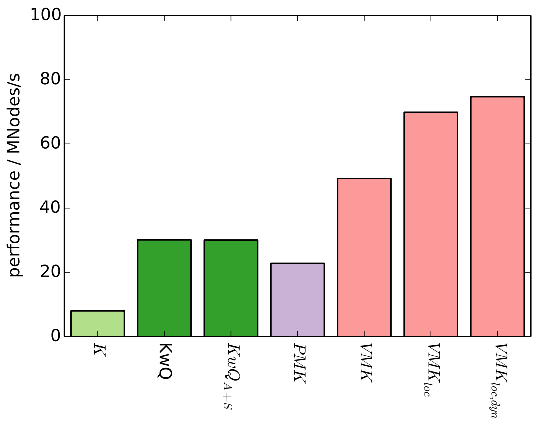
(e) GTX TITAN Stay

(f) GTX TITAN Even

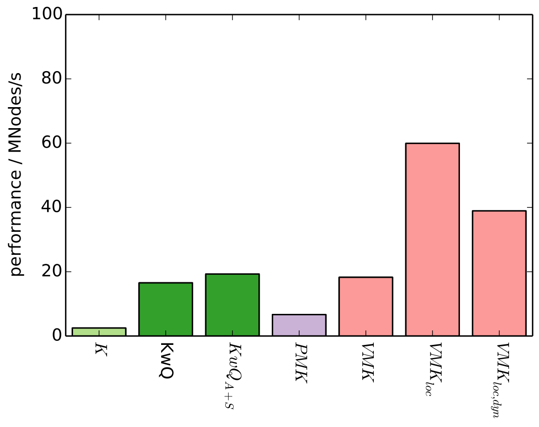
Figure 6.6: Fully filled tree on a GTX 580, 680, and TITAN. Performance given in MNodes per second.



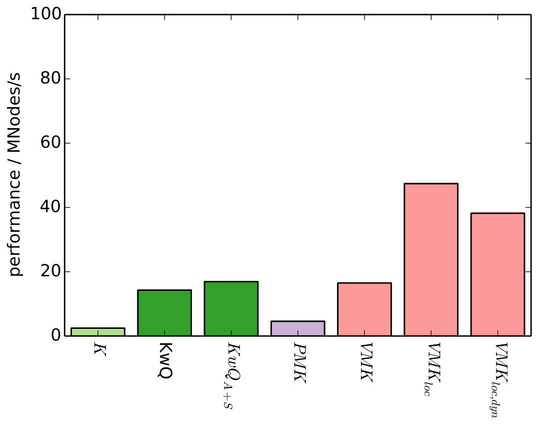
(a) GTX 580 Sparse Stay



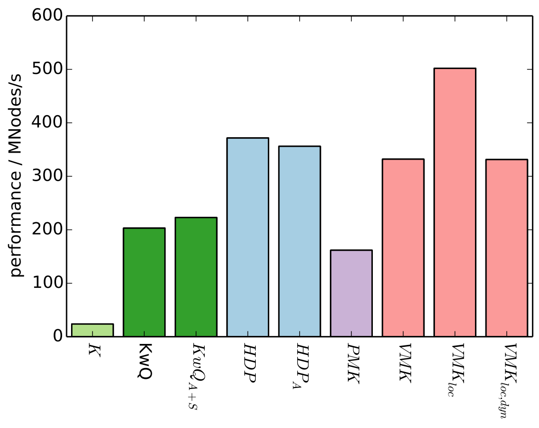
(b) GTX 580 Sparse Change



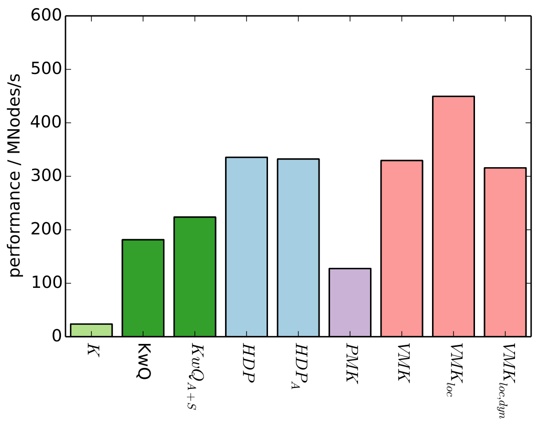
(c) GTX 680 Sparse Stay



(d) GTX 680 Sparse Change



(e) GTX TITAN Sparse Stay



(f) GTX TITAN Sparse Change

Figure 6.7: Sparse forest on a GTX 580, 680 and TITAN - MNodes/s.

Figure 6.7 shows the maximum number of executed tree nodes per second for the sparse forest scenarios. We see a similar picture. K does not work well. KwQ performs reasonable well, but is clearly outperformed by VMK. HDP in this case clearly outperforms KwQ. PMK again falls short of VMK due to divergence. VMK outperforms the other approaches using queues in shared memory by up to 300%. Although 60 thousand nodes form a reasonable load for a GPU, the overhead of reading data back to the CPU seems to be too high for 500 traversal steps, *i.e.* 2000 kernel launches, thus HDP in this case can also outperform KwQ clearly. For VMK, queuing in shared memory works better than global queuing on all devices. Interestingly, on the GTX 580 the dynamic shared queues, perform better than the fixed size shared queues. Although the dynamic queues introduce a serious management overhead, they offer the ability to dynamically assign the available shared memory to different procedures. While this approach also works well on the GTX 680 (only small overhead in comparison to VMK_{loc}, it does not generate in performance gain over the global queues on the GTX TITAN. Overall the performance on the GTX TITAN is more than five times higher than on the GTX 580 and 680. The 680 performs significantly worse than the 580. We did not find a significant difference between the *stay* and *change* scenario for this setup.

6.5.4 Pipeline

The most complex synthetic test is the pipeline test. This test evaluates the systems ability to cope with varying attributes of the individual procedures. Overall, we generate 32 different pipeline scenarios, which are based on the combination of varying the following attributes:

- number of stages: 4 vs 7
- data size: 16 vs 78 bytes
- type of input data: packages only vs item sets and packages
- thread count: equal vs varying (256 for packages and 1 for items vs 64-512 and 1-16)
- register count: equal vs varying (32 register vs 8-63 register)

In addition, we varied the synthetic load of each stage between 20 and 1800 FMADs. We also vary the probability of generating an element for the next stage adjusted to the number of threads used in each procedure. Overall we tried to build pipelines with expanding behavior, overall generating work for more threads than were used in the current stage. We started each pipeline with 3000 initial packages. To keep queue lengths as short as possible we preferably draw elements from the back of the pipeline. As QwK and HDP do not offer the ability to prioritize different procedures, there is no control over queue lengths, which leads to maximum fill rates of 2 million items. The megakernel approaches (especially with local queuing) require considerable less queues space, with a maximum fill rate of 100 thousand elements. PMK are not included in the test as they are not able to mix different procedure granularities and could thus only execute two pipelines.

A subset of all execution times is shown in Table 6.3 and the entire results can be found in Appendix D. The tables include an *unscheduled* version, which only executes

Table 6.3: Relative execution time for the 7 stage pipeline test with FMADs as artificial workload and small data on a GTX 580, 680 and TITAN.
et = equal threads; vt = varying threads, er = equal registers, vr = varying registers

(a) GTX 580

	Packages				Mixed			
	et		vt		et		vt	
	er	vr	er	vr	er	vr	er	vr
Unscheduled	0.83	0.83	0.84	0.84	0.02	0.03	0.05	0.04
K	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
KwQ	1.00	1.01	**0.99**	0.99	0.40	0.48	0.50	0.66
KwQ_{A}	0.96	0.99	1.02	1.07	0.33	0.44	0.65	0.79
KwQ_{A+S}	1.01	1.00	1.00	1.03	0.36	0.44	0.66	0.80
VMK_{B}	1.03	1.06	1.95	2.18	0.31	0.40	0.87	0.95
$VMK_{B,shared}$	**0.89**	**0.84**	1.14	1.48	0.22	0.25	0.56	**0.51**
VMK	1.04	1.10	1.29	1.34	0.31	0.38	0.76	0.78
VMK_{shared}	0.90	0.85	1.07	0.93	0.20	0.26	**0.50**	0.59
$VMK_{shared,out}$	0.89	0.85	1.08	**0.91**	**0.19**	**0.24**	0.64	0.66
$VMK_{shared,dyn}$	1.14	1.18	1.54	1.57	0.35	0.45	0.94	1.24

(b) GTX 680

	Packages				Mixed			
	et		vt		et		vt	
	er	vr	er	vr	er	vr	er	vr
Unscheduled	0.34	0.27	0.26	0.24	0.01	0.01	0.01	0.01
K	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
KwQ	1.07	1.04	0.71	0.73	1.16	0.94	0.90	0.79
KwQ_{A}	1.03	1.03	0.72	0.71	1.33	0.83	0.86	0.70
KwQ_{A+S}	1.03	1.02	0.72	0.71	1.19	0.84	0.90	0.70
VMK_{B}	2.48	2.10	2.07	2.17	1.07	0.69	1.01	0.84
$VMK_{B,shared}$	**0.34**	0.29	0.47	0.47	0.46	0.39	0.65	**0.50**
VMK	2.18	2.22	1.12	1.15	1.05	0.71	0.86	0.82
VMK_{shared}	0.35	**0.29**	**0.28**	**0.26**	0.51	**0.37**	**0.62**	0.54
$VMK_{shared,out}$	0.36	0.32	0.29	0.26	**0.44**	0.39	0.74	0.54
$VMK_{shared,dyn}$	0.47	0.49	0.81	0.83	0.62	0.45	1.00	0.52

Table 6.3: Relative execution time for the 7 stage pipeline test with FMADs as artificial workload and small data on a GTX 580, 680 and TITAN.
et = equal threads; vt = varying threads, er = equal registers, vr = varying registers

(c) GTX TITAN

	Packages				Mixed			
	et		vt		et		vt	
	er	vr	er	vr	er	vr	er	vr
Unscheduled	0.86	0.76	0.78	0.81	0.04	0.04	0.05	0.05
K	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
KwQ	0.97	1.00	0.99	0.99	0.37	0.39	0.63	0.75
KwQ_A	0.96	0.99	0.93	0.99	**0.26**	0.38	0.56	0.77
KwQ_{A+S}	0.94	0.98	0.92	0.99	0.28	0.39	**0.55**	0.78
DP	19.09	20.57	18.26	22.79	n.a.	n.a.	n.a.	n.a.
HDP	0.91	0.88	0.95	1.00	0.48	0.37	n.a.	n.a.
HDP_A	**0.90**	0.88	0.97	1.01	0.54	0.43	n.a.	n.a.
VMK_B	1.00	0.99	1.86	2.13	0.41	0.44	1.00	0.97
$VMK_{B,shared}$	0.91	0.78	1.38	1.49	0.28	0.34	0.74	0.79
VMK	1.00	0.98	1.06	1.16	0.37	0.44	0.98	0.99
VMK_{shared}	0.91	**0.78**	**0.81**	0.84	0.29	0.35	0.71	**0.71**
$VMK_{shared,out}$	0.91	0.79	0.82	**0.83**	0.27	**0.31**	0.70	0.84
$VMK_{shared,dyn}$	1.01	0.96	1.17	1.31	0.43	0.51	1.16	1.36

the synthetic load, as if there was no scheduling and memory movement required. The performance of K and KwQ is in general good in this example. Because the pipeline is strictly feed-forward, there are only six control switches required between CPU and GPU and there is only one kernel launched in each iteration. Thus, there is only a small overhead associated with kernel launches. Additionally, it explains why KwQ_{A+S} does not show any benefit over KwQ_A. As expected, K perform well for package only data on all devices. For mixed granularity pipelines, KwQ outperformed K due to its ability to schedule multiple items and item sets together. KwQ (and its variants) achieved the best overall performance in 3 out of 24 cases. On the GTX TITAN HDP_A achieved the best performance for packages with equal threads and equal registers. HDP in general was on par or even slightly faster than KwQ. The access variants $_A$ in general increased performance slightly over copying the data to shared memory. DP performed very badly in comparison to K, resulting in slowdowns of about 20 times.

The best overall performance was again achieved by VMK with its variants, taking the lead in 20 of 24 cases. The variants scoring the highest score were always based on queues in share memory. The global variants could hardly increase performance compared to KwQ and HDP on any device. For varying thread counts, especially in the package scenario, the VMK_B which does not offer the ability to execute multiple packages concurrently dropped by a factor of 1.6 on average (580: 1.7, 680: 1.2, TITAN: 1.9) The full VMK approaches did not suffer from this loss and, on average, reduced the execution time or kept it on the same level compared to the base line (580: 1.2, 680: 0.7, TITAN: 0.9) In the mixed scenario the item procedures introduce the biggest overhead and there are only three package procedures with thread count 128, 256, and 384. Thus, there is less opportunity for performance gain. Still, the when comparing the global queuing approaches VMK_B and VMK an average gain of 10 to 15% is visible. In two cases, the approaches using shared memory queuing, show the opposite behavior. If no packages are executed concurrently, fewer new procedures are generated. Thus, the shared queues are less likely to run out of space and the procedures stored in shared memory can be executed before new procedures are being generated.

In this test, we also compared different strategies for assigning queue space to procedures. In 15 of 24 cases, assigning the same amount of shared memory to all procedures worked best. In 9 cases, assigning more space to the latter stages of the pipeline increased the performance. On average, using equal amount of shared memory worked 1 to 2% better than having more space for the final stages available. Dynamically assigning the queue space to the procedures as needed did not achieve the best performance in any case and on average was between 20 and 30% slower.

Besides the reported difference of VMK_B and VMK, varying the thread count only slightly reduced the performance of the megakernel approaches (relative to KwQ_A). On all devices the varied register count did not have a serve influence on the performance of the megakernel approaches. Surprisingly, the relative performance was even higher for varying registers than for equal registers in some cases.

Overall, we recorded a small scheduling overhead for package data. The *unscheduled* baseline version is only a few percent faster than the fastest scheduled version. For the mixed scenario, which generates millions of intermediate items, the overhead of memory movement is more severe. On the GTX 580 and TITAN the overhead reduces performance by about 8 to 10 times. This indicates that memory access and scheduling makes up for the majority of execution time if one needs to schedule individual work for millions of threads with little workload. On the GTX 680 the performance is even reduced by 30 to 50 times, again indicating that the GTX 680 is mostly tuned for pure floating point performance.

6.6 Discussion

Our experiments have shown that software-based scheduling of diverse tasks is feasible on the GPU. The traditional host-controlled kernels and the recently proposed dynamic parallelism do not provide the capabilities to efficiently schedule tasks with arbitrary granularities of parallelism. Our proposed variants which use efficient queuing strategies not only provide these functionalities, but also outperform the traditional interfaces in scenarios that fit the kernel-based model well.

Our extension for the kernel-based execution model consists of individual queues for each procedure (KwQ) and uses individual kernel launches for all queues. This approach provides good performance if the number of required kernel launches is low in comparison to the number of elements in the queue. While dynamic parallelism used in a straight forward manner performed poorly in all scenarios, our hybrid dynamic parallelism (HDP) implementation showed a competitive performance. In the tree and pipeline test, HDP was able to outperform the KwQ approach. For the recursive test case, especially when there was a severe load on the memory bus, HDP fell clearly behind. When optimizations are applied to all our approaches the difference between KwQ, and HDP, VMK is most often below 25% on the newest GPU architectures.

Especially our versatile megakernel (VMK) showed a very promising performance. While previous megakernel approaches did not support the execution of different granularities of parallelism and had problems with divergence, we could solve these shortcomings with our VMK. Using different queues for individual procedures solves the problem of divergence. Building on inline PTX code to provide multiple synchronization barriers, we are able to dynamically assign threads to procedures and, in this way, support the execution of arbitrary thread groups that can work cooperatively. By choosing the megakernel block size based on the threads required by the individual procedures, we could significantly boost performance. In the simple recursive scenario, VMK achieved the best performance in 19 of 24 test cases. When traversing a fully filled binary tree, VMK variants dominated the performance in 90% of all cases. In the sparse forest scenario, VMK always achieved the best performance. When executing our seven stage pipeline with strongly diverse procedure characteristics, VMK was the fastest approach in 20 of 24 cases.

The biggest advantage of VMK is its ability to use queues in on-chip shared memory, while all other approaches need to go through global memory. Using shared memory can boost performance of up to 100% on the GTX 580 and GTX TITAN and up to 500% on the GTX 680. The downside of megakernel approaches is the potential performance loss introduced by compiling all procedures into a single kernel, reducing the achievable occupancy for diverse procedures. Interestingly, varying thread counts did not have a strong impact on performance. In some cases VMK even performed better for varying register counts.

When comparing different devices, we saw that there is only a small difference between the best performances of the three generations of graphics cards for the recursive test. While the GTX 580 and 680 achieve between 60 and 70% of peek performance, the GTX TITAN only achieves about 50%. Note that the GTX 680 strongly benefits from queuing in shared memory. In the tree and forest examples the GTX TITAN shows its strengths with up to 5-10 times more performance than the other devices. The GTX 680 falls short of the GTX 580, being up to 5 times slower than the 580. In the pipeline scenario the devices were again closer in terms of performance for the package only scenario, whereas the GT 580 was the slowest device, followed by the GTX 680 and the GTX TITAN. In the mixed scenario, the GTX 680 performed worse than the 580, being three times slower. Overall the scheduling and memory transaction overhead for the pipeline scenario was low for package data, but high for the mixed scenario. The comparison between the different graphics cards shows that the GTX 680 was mainly designed for good floating point performance but has problems, when there are many memory accesses, logics, or atomic operations involved. The GTX TITAN achieves the highest performance overall, but it seems to be harder to get closer to peak performance, even if there is sufficient instruction level parallelism.

Overall, we clearly recommend VMK for task scheduling on the GPU. It not only provides the best performance, but VMK is the most flexible approach in terms of scheduling policies. Based on the way the queues are organized and from which queue data is drawn first, the overall scheduling can be influenced. Different tasks can be prioritized in different situation, to take influence on queue fill-rates or execute more important tasks first. Based on the already presented results, we see a strong indication that our VMK approach is able to execute dynamic algorithms with high performance on current graphics hardware (**O1**). Still, we show various real-world examples in the final part of the thesis confirming this claim.

The performance of our proposed methods KwQ, HDP, and VMK indicates that an efficient scheduling of tasks with varying granularities of parallelism is possible on graphics processors (**O2**). Our approaches not only outperform the basic dynamic parallelism implementation by at least one magnitude, they operate efficiently in cases where dynamic parallelism fails. To fully validate **O2**, we will show that a real-time Reyes pipeline with different degrees of parallelism and high shared memory requirements can be build with our approach (Section 8).

Chapter 7

Dynamic Memory Management

Contents

Dynamic memory allocation is an indispensable feature of modern operating systems and virtually every computer program depends on this feature to allow for a dynamic response to varying inputs. Consequently, almost every general purpose programming language provides some mechanism to create new objects at runtime. Efficient dynamic memory allocation became more demanding with the advent of multi-core CPUs. Since system memory is a shared resource, conflicts during memory allocation are an issue. To avoid conflicts, synchronization among multiple cores is required. However, this synchronization requirement defines a serious bottleneck for many applications [98]. For GPU architectures, dynamic memory allocation is an even greater challenge. While a current consumer CPU features between four to eight cores, current graphics cards are shipped with thousands of cores. In addition to the much larger number of cores competing for memory, the architecture's high latency for accessing global memory complicates the development of an efficient dynamic memory allocator.

Since two generations of GPU architectures, the problem of high latency for global memory access has been partially alleviated due to the introduction of a cache [100]. For this architecture, NVIDIA's CUDA model supports dynamic memory allocation in device code [103]. Dynamic memory allocation for the stream programming model opens up completely new possibilities for GPU programming.

Examples for an efficient use of this new feature range from image detectors, which store local descriptors of varying size, over parallel IP-packet stream analysis, which dynamically generates alerts, to weather simulations, for which pressure area descriptors are dynamically created. Previously, organizing vast numbers of differently sized objects at runtime was only possible using parallel reduction summations (*scan*) [37]. The *scan*-method requires multiple kernel launches: One initial kernel launch to analyze the input data and write the number of required bytes for each data element into global memory, followed by the scan kernels. Subsequently, a final kernel launch is required to write the resulting data.

Compared to this procedure, a single call of a dynamic memory allocator function is very likely to be more efficient in terms of programming effort and execution time.

Although NVIDIA makes their dynamic memory allocator available for use within the CUDA toolkit, its internals remain undisclosed. Furthermore, our tests have revealed that their current implementation is unreliable under heavy load. Literature on dynamic memory allocation on massively parallel architectures is rare. The only published and especially for the use on the GPU optimized dynamic memory allocator is *XMalloc* [48]. Therefore, we see an urgent need for further research in this area. With this work, we want to form a solid base for the design of dynamic memory allocation on massively parallel architectures such as the GPU. To achieve this goal, we design and implement *ScatterAlloc* and give detailed information about its inner structure, performance and implementation.

7.1 Design Goals

The general design goals for dynamic memory allocators are architecture-independent: **correctness**, **speed** and **little memory consumption**. To design an allocator for the GPU, we want to build on the know-how from the field of multi-core CPU memory allocation. In the following, we detail on the special requirements and distinct features that have to be considered for the GPU.

7.1.1 Correctness

A fundamental requirement for every deterministic algorithm is of course its correctness. A primary issue concerning the design of a GPU allocator is that it is virtually impossible to make any assumptions about scheduling on the GPU. This means that mostly lock-free algorithms have to be used to avoid deadlocks. Consequently, many well known CPU memory allocation methods drop out at this stage.

7.1.2 Speed

Keeping the number of clock cycles spent with allocation and deallocation of memory low, must be a major goal of allocators for the GPU. However, not only the time spent on the allocation and deallocation is important, but also how efficiently the allocated memory can be accessed.

Memory-access performance One important property of current the GPU architectures is the fact that memory access can be extremely costly in relation to computations [20, 78]. On an Intel Core i7, data access takes about 4, 10, and 38 cycles, for data in L1, L2, and L3 cache respectively and about 100 cycles for accessing data in main memory [94]. On the Fermi architecture, access times are, according to our own measurements, about 18, 250, and 1000 cycles for data in L1 cache, L2 cache, and global shared memory respectively. Additionally the cache sizes on the GPU are about one tenth of the cache sizes on a CPU, which becomes even more severe considering that the L2 cache on a GPU has to serve hundreds of cores and thousands of threads. Therefore, *ScatterAlloc* keeps the number of data accesses low, while it can spend more time on complex computations.

Scalability A lock-free allocator usually relies on atomic operations to handle concurrent allocations of multiple threads. The time it takes for such an operation to complete is proportional to the number of threads accessing the same data word in parallel. Although it is believed that a linear performance decrease is a good scalability for CPU-based allocators [8], it cannot be acceptable for GPU execution, where thousands of threads execute concurrently. Our measurements show that, for multiple threads atomically accessing the same data word, the average access time increases with at least 100 cycles per thread. Consequently, atomic operations on the same data word are avoided whenever possible by *ScatterAlloc*. Ultimately, a perfect allocator should be unaffected by the number of threads concurrently allocating data and thus provide allocation and deallocation in constant time.

Diverging execution paths As the GPU is based on the SIMD model, warps can only execute in a coherent manner. This means, that if multiple threads within a warp concurrently allocate memory, the best performance will be reached, if they all execute the same code. Diverging branches are synchronized by the warp scheduler, forcing all threads to wait for the whole warp to finish, which can strongly reduce performance.

False sharing Another important issue with CPU-based allocators is *false sharing*. *False sharing* means that data accessed by different processors should not be placed in the same cache line if the data are not intended to be shared. Consequently, data accessed by one processor should be strung together to speed up the data access. Since the introduction of the *Fermi* architecture, the GPU also caches global memory accesses, making *false sharing* an issue for efficient GPU programming. Because all threads executing on the same multiprocessor share the same cache, *false sharing* occurs between different multiprocessors only, while cache line sharing between different threads on the same multiprocessor can be intended.

Coalesced access Traditionally, data access on the GPU could only be performed efficiently if all threads of a block accessed the data words linearly according to their thread index, with no gaps in between, hence, in a coalesced manner. Although this is not necessary for current graphics cards anymore, data words accessed by adjacent threads can be read with a single load instruction, if they are placed close to each other in memory. If threads of the same block allocate data at the same point in time, it is likely that these data words will also be read or written concurrently. Thus, *ScatterAlloc* considers it important that data words allocated by threads of the same block at the same time are close to each other in memory.

7.1.3 Memory consumption

The lower the overall memory consumption of an allocator, the less data has to be accessed, the more data fits into the cache, and the more memory is free to be used by the system. Thus, a major design goal of dynamic memory allocators is to reduce the overall memory consumption. This includes memory fragmentation, which is a measure of the unusable regions between occupied memory, and furthermore the overhead of the data structures that are used to keep track of free and used regions. To be able to formalize our measures of fragmentation, we build on the assumption that memory is split up into regions. We define the set of all regions R as well as the functions $alloc : R \rightarrow \mathbb{N}$ and $size : R \rightarrow \mathbb{N}$. $alloc(r)$ maps a region to the size of the memory request it has been allocated for, or 0 if it is free. $size(r)$ gives the actual size of a region. Based on these functions, we can define the set of all allocated regions $A = \{r \in R | alloc(r) \neq 0\}$ and the set of all free regions $F = R \setminus A$.

Internal fragmentation Internal fragmentation occurs, if the size of the allocated memory region $size(r)$ is larger than the requested size $alloc(r)$. This may be necessary to meet alignment requirements of the processor, *e.g.*, for the Fermi architecture objects must be 16 byte aligned, or due to the internal structures of the allocator. Internal fragmentation may also vary with the allocation pattern of the application. We compute the internal fragmentation $F_{internal}$ as the average of relative wasted space among all allocated memory regions:

$$F_{internal} = \frac{1}{|A|} \cdot \sum_{r \in A} \frac{size(r) - alloc(r)}{size(r)}. \tag{7.1}$$

External fragmentation External fragmentation occurs, if the available free memory is divided into small chunks, which might be too small for direct use by the application. The amount of external fragmentation is strongly dominated by the allocator's strategy for finding free memory regions. Its value $F_{external}$ is commonly defined as the ratio of the largest free memory region to overall free memory:

$$F_{external} = 1 - \frac{\max_{f \in F} size(f)}{\sum_{r \in F} size(r)} \tag{7.2}$$

Consequently, an external fragmentation of 0 means that all available memory can be allocated in one big chunk.

Blowup *Blowup* occurs if the allocator's memory requirements disproportionally increase over time compared to the amount of allocated memory. The reason for this behavior can only be found by analyzing the internal structure of an allocator. In most cases, *blowup* is caused by the disability to reallocate previously freed memory. *Blowup* can dramatically increase memory usage for certain scenarios and is therefore considered in the design of *ScatterAlloc*.

7.2 A Parallel List-Based Allocator

To obtain a baseline for the comparison with *ScatterAlloc*, we have implemented a traditional allocation scheme based on a *first-fit* algorithm on the GPU. Memory is organized in consecutive segments that each keep pointers to the previous and next segment. We refer to this structure as the *segment-list*. A memory segment can either be allocated or free. When a request is made for a new block of memory, the list of segments is traversed until a free segment of suitable size is found. If the size of the first fitting segment is larger than requested, the segment is split such that one segment then is of the appropriate size for the memory request and the other one contains the remaining memory. The first of the two parts is subsequently marked as allocated and returned to the caller.

To speed up the search process, we maintain a second list of free segments so that only those are considered during the search for a free block of memory. When a segment is allocated, it is removed from this *free-list* and if a split yields a free segment, this segment is inserted back into the list. To counter external fragmentation, the allocator attempts to merge segments back together with their immediate successor within the segment-list on deallocation.

For CPU allocators it is common to organize multiple heaps of smaller size to reduce allocation time. This is well demonstrated in numerous previous attempts [8, 23, 71, 91, 121, 123]. In an approach similar to *skiplists* [112], we introduce a partial ordering on the elements of the free-list, organizing it into sub-lists that contain blocks of certain sizes. A number of what we call bins each store entry points to these sublists. The way these bins are created and managed allows the implementation of various allocation strategies, including strategies similar to the use of multiple smaller heaps.

To build these data structures, management information has to be associated with every memory block. This information consists of a bit-field defining the current state of the block, as well as pointers to the previous and next blocks in both, the segment-list and the free-list. We store this information right before the actual memory block. Dealing with a highly parallel architecture, we want to support concurrent insertion and deletion of list elements. Thus, access to these pointers needs to be synchronized to avoid corruption.

Synchronization is also needed during the allocation process because multiple threads might try to allocate the same block of memory. If a thread identifies a suitable block of memory, it atomically tries to set the allocated flag. If this is successful, it is free to use the entire block or split it into two blocks.

With this functionality at hand, we have first created an allocator similar to dlmalloc [71] by inserting free memory blocks into the sublists according to their sizes. This enables a thread to directly jump to a first element that potentially satisfies its demands, instead of having to walk the entire free-list. As the individual sublists remain connected as a whole, a thread that has not been able to secure a block from its initial sublist will automatically run over to the next bin, splitting up a bigger block of memory.

For a second implementation we create a fixed number of bins and assign threads according to their execution locality to these bins. This creates a behavior similar to the Hoard system [8] and the ‘Scalable lock-free dynamic memory allocation’-scheme [91]. We can assign all threads running on a single multi processor to the same bin, which has a positive impact on the number of cache hits and reduces congestion.

However, all these methods are not optimal for use on highly parallel architectures like the GPU, as mentioned previously and confirmed by our results in Section 7.5. In the next section we therefore propose the novel design of *ScatterAlloc* that overcomes the inherent problems of the methods presented here.

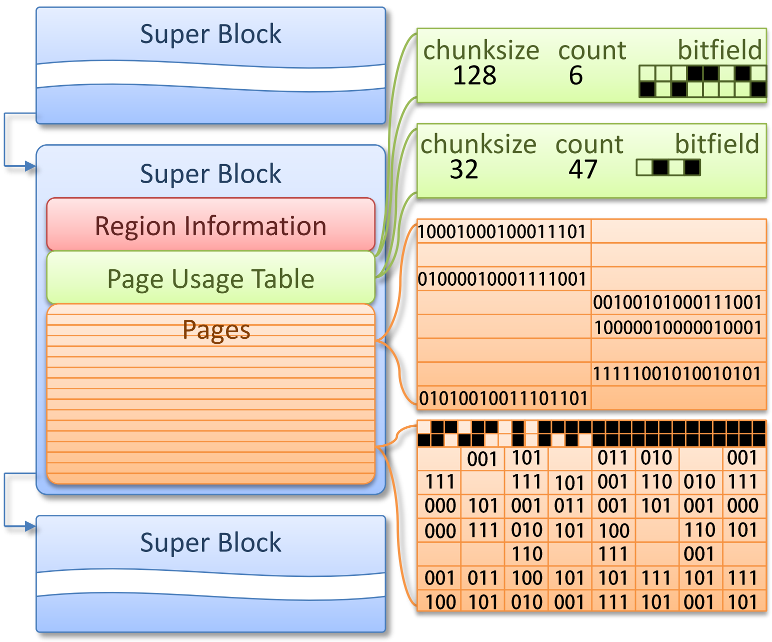


Figure 7.1: Overview of the data structures used in the *ScatterAlloc*: Memory is organized in super blocks which are collected in a list. Each super block holds a fixed number of equally sized pages, information about the usage of each page and meta information collected for regions of pages. The *usage table* captures the size according to which the page is split into equally sized chunks, the counter of used chunks and a bit field to identify the used chunks. To allow for splitting the page into more chunks than actually can be captured by the bit field in the usage table, an additional hierarchy of bits is placed on the beginning of the page.

7.3 ScatterAlloc

Traditional allocation strategies are too slow to be used where a large number of threads concurrently allocate memory. In an environment where hundreds of threads try to seize the first block of memory that fits their needs concurrently, collisions quickly become the major bottleneck of an allocator. To avoid these collisions, *ScatterAlloc* scatters allocation requests across fixed sized memory regions, trading fragmentation for allocation speed.

ScatterAlloc organizes its memory by splitting it into fixed sized *pages*. To keep track of the free memory within a page, we employ a *page usage table*. Pages are grouped together in *super blocks*, which form the biggest unit of memory for *ScatterAlloc*. To speed-up the search for allocable memory, we store meta data about the fill-level of different regions within each super block. This meta data is designed to be frequently read but rarely written in order to reduce congestion. For an overview of the data structures and their relations, see Figure 7.1.

7.3.1 Super blocks

On current NVIDIA CUDA devices, a fixed size memory pool is used to serve dynamic memory requests. The pool's size has to be set before a module is loaded into the context. Thus, it would currently be sufficient to design an allocator which can manage a memory region of a fixed size. As this behavior might change in future, we design our allocator in such a way that it can either be used on one big region of memory, or manage a variable number of memory blocks which can be found at arbitrary locations in memory. This enables *ScatterAlloc* to be applicable on future architectures, which might be able to increase the size of the heap during run-time. It is also possible to dynamically add memory to the pool manged by *ScatterAlloc* on current architectures. In case the allocator is almost out of memory, a host side *alloc* between two kernel launches can be used to allocate a new region of memory which is then passed to the allocator before the next kernel launch is executed. According to the common terms used for CPU allocators, we call these coherent regions of memory assigned to the allocator *super blocks*. We require all super blocks to be of equal size. If the allocator is configured to manage a single large region of memory, we split it up into multiple super blocks. To allow super blocks to be scattered in memory, we organize them in a singly linked list.

7.3.2 Pages

To facilitate parallel, collision-free memory allocation, we split every super block into fixed sized pages. This way, there is no need to search through lists when allocating memory because every thread can calculate the page offset from the size and address of the super block. To serve memory requests, pages are split into equally sized chunks. The chunksize on a page is set during the first access to the page. Once this split has occurred, it remains fixed until all chunks allocated on the page have been freed again. This strategy clearly favors the allocation of small and similarly sized chunks of memory. But this is actually the behavior we expect from a program using dynamic memory allocation in a highly parallel fashion: there will be multiple threads dealing with similar inputs, requesting similar amounts of memory. At the same time, these memory requests have to be small in relation to the entire available memory, because otherwise the system would run out of memory immediately.

To keep track of the chunks within a page, we facilitate a *page usage table*. Every entry in the page usage table consists of three values: the chunk size, the number of allocated chunks, and a bit-field. In this bit-field, each bit represents a single chunk of memory. A high bit means that the associated chunk of memory is in use, while a low bit indicates a free chunk. We require atomic operations and fast bit operators on this bit-field for *ScatterAlloc* to work efficiently. However, atomic operations are generally only supported for 32 or 64 bit words, with bitwise atomic operations only available on 32 bit words. Hence, the bit-field is 32 bit long in our current implementation. To support splitting a page into more than 32 chunks of memory and to handle smaller memory requests without a high amount of internal fragmentation, we introduce a second hierarchy level: On the page itself, we place up to 32 additional bit-fields, one linked to each bit in the page usage table. This way, we support splitting a page into up to $32^2 = 1024$ chunks.

7.3.3 Hashing

The key ability for a fast allocator on the GPU is to avoid collisions. If multiple threads try to allocate the same chunk of memory, performance can drop quickly. To avoid collisions and to quickly find suitable pages for allocation, we rely on hashing. While previous approaches use only the thread identifier for their hash function to determine a local heap, *e.g.* proposed by Larson *et al.* [69], our hash function fulfills additional requirements:

- When yielding a page, it seeks to split the page into chunks of the same size as the memory request. In this way, a free chunk on the page can be used and internal fragmentation will stay low.
- Threads running on the same multiprocessor aim for allocating memory side by side so that the cache utilization is higher.
- For our implementation in CUDA, threads within the same warp aspire to allocate their data adjoined as possible, as they will likely access data coherently.

To build the hash-function, we use the requested memory size and information about the location of the threads' execution (multiprocessor id). Using multiplicative hashing [59] we control the influence of the two factors, when determining a suitable page p:

$$p = (S_{requested} \cdot k_S + \mathrm{mp} \cdot k_{\mathrm{mp}}) \mod \mathrm{SP}, \tag{7.3}$$

where $S_{requested}$ represents the requested memory size, mp stands for the multiprocessor id, and SP denotes the number of pages within the super block. The factors k_S and k_{mp} can be used to distribute the access across the entire super block or to determine a local offset for the data. If we choose k_S to be a large, possibly a prime number, it is more likely that pages will be found which have been split with the same chunk size. Thus, internal fragmentation is kept low. A small value for k_S will cause little scattering of similarly sized memory requests. In combination with a large k_{mp}, data allocated by the same multiprocessor will be allocated close to each other. Thus, cache hits are more likely. However, in this case differently sized chunks will be placed on the same page, increasing internal fragmentation. The optimal choice depends on the usage pattern of the application.

In case the determined page is already full, we search for a free chunk in subsequent pages, which will introduce local clustering. This strategy results in better cache-utilization when searching for free pages, while the aforementioned considerations of either well matching chunk size or similar multiprocessor id also apply.

7.3.4 Meta Data

To speed up the search for free suitable chunks, we introduce a two level hierarchy of meta data. Meta data is read during each memory request, while it is only written occasionally. Our allocator always keeps a pointer to the currently active super block. Only if the super block reaches a certain fill level, or a memory request cannot be handled by the super block, the allocator proceeds to the next super block in the list. As a thread proceeds to the next super block, it updates the active super block pointer.

To quickly reject memory regions which are unlikely to contain suitable free chunks, we divide each super block into equally sized regions. For every region, we keep information about how many pages are full. If an allocation request fills up one page, it increases the counter for the region the page lies in. If a region is about to run out of free pages, memory requests are forwarded to the next region. As it takes some time for threads to report that pages are full, we already proceed to the next region when 90% of the pages are full. If a chunk of a full page is freed, the region counter is decreased, keeping the meta information accurate.

The available space within a region plays an important role in reactivating super blocks. Threads which free a chunk on a non-active super block randomly choose to set this super block as the active super block. The likelihood that this update is performed increases with decreasing fill-level. This helps to avoid memory blowup, as it allows for super blocks to be reused.

7.4 Implementation

To discuss the implementation details of our allocation strategy, we provide pseudo code of *ScatterAlloc*'s allocation and free methods. We require the target architecture to support atomic operations on global memory and efficient bit operators, as, *e.g.*, provided by the current CUDA C programming language.

7.4.1 Alloc

The pseudo code for *ScatterAlloc*'s allocation algorithm is split up in two functions: *alloc* and *tryUsePage*. Note that we omit the steps for placing more than 32 chunks on a single page, as they are very similar to placing only 32 chunks per page. The only difference is another hierarchical level, for which the bits in the page usage table are used to indicate that all associated 32 chunks of the second hierarchy are in use.

To allow placing smaller requests in bigger chunks, we introduce a multiplicative factor `max_frag` (see alloc line 3) to control how much space is allowed to be left empty between two adjacent chunks on a page. This factor is directly related to the maximum internal fragmentation.

While searching for a suitable page, three situations can occur:

- If the page's chunk size is either too small or too big to be used for the current request, we move on to the next page (alloc line 31).
- If the page's chunk size fits the current request, we try to use the page (alloc line 11-14).
- If the page is not yet in use, the page's chunk size is marked as zero. In this case, we use an atomic compare-and-swap with the goal to set the chunk size of the page according to our needs (alloc line 17-26). In case no other thread has set it before, the page can be used for allocation. If another thread set the chunk size in the meanwhile, it is still possible that the set chunk size fits the current request.

If *ScatterAlloc* identifies a page to be used for the current request, we first increase the fill level of the page to see if there is space for another chunk (tryUsePage line 2). In this way, we check if there is an available spot on the page concurrently with reserving a

Function alloc(Superblock,S_{req})

```
1  p ← hash($S_{req}$,mp)
3  $S_{\max}$ ← $S_{req}$ · max_frag
4  while tried not all regions do
5      region ← p/ regionsize
6      if region filllevel ≤ regionsize ·9/10 then
7          while p is in region do
8              page ← Superblock →p
9              $S_{\mathsf{chunk}}$ ← page →chunksize
11             if $S_{\mathsf{chunk}} \geq S_{req}$ and $S_{\mathsf{chunk}} \leq S_{\max}$ then
12                 loc ← tryUsePage(page, $S_{\mathsf{chunk}}$)
14                 if loc ≠ 0 then return
15                 loc
17             else if $S_{\mathsf{chunk}} = 0$ then
                   // page is free so try setting $S_{\mathsf{chunk}}$
18                 $S_{\mathsf{chunk}}$ ← atomCAS(page →chunksize, 0, $S_{req}$)
19                 if $S_{\mathsf{chunk}} = 0$ then
                       // use the new page
20                     loc ← tryUsePage(page, $S_{req}$)
21                     if loc ≠ 0 then return
22                     loc
23                 else if $S_{\mathsf{chunk}} \geq S_{req}$ and $S_{\mathsf{chunk}} \leq smax$ then
                       // someone else acquired the page,
                       // but we can also use it
24                     loc ← tryUsePage(page, $S_{\mathsf{chunk}}$)
26                     if loc ≠ 0 then return
27                     loc
28                 end
29             end
31             p ← p + 1
32         end
33     else
           // try next region instead
34         p ← (region +1)·regionsize;
35     end
36 end
37 return 0
```

spot. After this step has been performed successfully, it is certain that the request will be served on that page. Without this counter, multiple threads might fight for the same spot. Note that this is the only point of serialization which depends on the number of threads trying to allocate memory concurrently. Nevertheless, not many threads will be directed to the same page as the hash-function will scatter their requests to multiple pages and full sections will be avoided due to the meta data structure.

After increasing the fill-level for the page, *ScatterAlloc* needs to determine which spot should be used. To do that efficiently, we use the bit-field of the page usage table (tryUsePage line 9-21). We start by estimating a free spot using the thread's lane id (thread index within its warp). This strategy is a good choice, because multiple threads of the same warp allocating equally sized data will be forwarded to the same page. In case all chunks are free, accessing them will result in coalesced memory accesses. If a chunk is already in use, we use the bit-field to quickly determine the next free spot as outlined in tryUsePage line 16-21. Essentially, we use bit-shifting in combination with CUDA's `__ffs` function to determine the closest free chunk on the page. Thus, we are able to find a free spot without the necessity to loop through the bit-field and mark the next free spot.

Function tryUsePage(page,S_{chunk})

```
 2 filllevel ← atomAdd(page →count, 1)
 3 spots ← calcChunksOnPage($S_{chunk}$)
 4 if filllevel < spots then
 5     if filllevel + 1 = spots then
 6         atomAdd(page →region, 1)
 7     end
 9     spot ← laneId % spots
10     while true do
11         mask ← (1 « spot)
12         old ← atomOr(page →bitmask, mask)
           // if the spot is free use it
13         if  old & mask = 0 then break
14
           // bit magic giving us the next free spot
16         mask = old » (spot + 1)
17         mask |= old « (spots - (spot + 1) )
18         mask &= (1 « spots) - 1
19         step ← __ffs( mask)
21         spot = (spot + step) % spots
22     end
23     return page →data + spot ·$S_{chunk}$
24 end
   // this page is full
25 atomSub(page →count, 1)
26 return 0
```

As long as our heuristic works, there will be few or no collisions and a suitable page will be found quickly. In this case, an allocation request can be handled within a minimum of five global memory operations (determining the active super block, checking the region status, comparing the page's chunk size, increasing the fill-level, and marking the spot). If multiple threads try to allocate data at the same time, most of the information will already be in the cache and the request can be handled faster. We also benefit from the cache in finding a free section and searching through the pages, as this data occupies a continuous region of memory.

7.4.2 Free

Freeing a chunk of memory is simpler than allocating, as demonstrated by the pseudo code for *free*. Given that we already know which super block the chunk belongs to, a few address and offset computations (free line 2-7) are sufficient to determine the bit that has to be cleared to mark the chunk as free and the counter that has to be decremented to remove the reservation of the spot on the page.

In case a page has become completely free, *ScatterAlloc* resets the way the page has been split by setting the chunk size in the page usage table to zero. To avoid race conditions in this situation, we have to lock the page, so that no other thread will try to allocate memory on this page with an incorrect chunk size. We implement this lock by atomically setting the page's fill level counter to a maximum if it is still free (free line 12). Now, if a different thread tries to reserve a spot on the page, it will fail and we can safely reset the chunk size of the page. This functionality also justifies using this counter instead of using the bit-fields only, which would not be sufficient to resolve the concurrency issue if secondary hierarchies of bit-fields on a page are used.

The provided pseudo code shows that two reads from global memory (data offset and chunk size) and two atomic operations on global memory are needed to free a chunk. The only source for congestion is formed by the two atomic operations. As the number of chunks on a page is limited, the bit-field operation will not cause a lot of congestion. The operation on the fill count may be subject to more congestion, as it is also atomically altered during the allocation process. Again, we can argue that the number of threads trying to request a chunk on a single page will still be low due to the use of a hash-function. Thus, the free operation is generally very efficient, as also shown in Section 7.5.

The only remaining question is how to determine the super block in which the chunk falls. In case of a single big allocator-managed block, a simple address calculation using one division is sufficient, because all super blocks lie next to each other in memory. When multiple separate super blocks should be used, a simple array with information about all available super blocks is all that is additionally required. While searching through this small array the L1 cache will be fully utilized and in an optimal case, the super block will be found about as fast as if we would only read a single value from global memory.

7.4.3 Large Data Requests

Up to now, we have described only requests that fit on a single page. In case a request is bigger than a page, we can serve this request by allocating multiple pages. As the used

Function free(SuperBlock,pointer)

```
p ← (pointer- Superblock →data)/pagesize
page ← Superblock →p
$S_{chunk}$ ← page →chunksize
spot ← (pointer - page →data)/$S_{chunk}$
mask ← (1 « spot)
// mark chunk free
atomAnd(page →bitmask, mask)
// reduce counter
count ← atomSub(page →count, 1)
if count = 1 then
    // this page now got free, try 'locking' it
    count = atomCAS(page →count, 0, pagesize)
    if count = 0 then
        page →chunksize ← 0
        __threadfence()
        // 'unlock' it
        atomSub(page →count, pagesize)
    end
end
```

memory is strongly scattered within a super block, we forward requests that are bigger than the page size to super blocks that are deliberately reserved for this purpose. In this super block, we search for a sufficient number of consecutive free pages. Then we try to reserve these pages by setting all chunk size fields of each page's usage table to the requested size. This will prohibit any other thread from using the page in further memory requests. If in the meantime another thread tries to acquire one of the pages that we want to use, we reset all already reserved pages and restart the search. Note that such big data requests will not be made by a large number of threads, as the system would run out of memory very quickly.

7.5 Evaluation

In this section we describe a set of tests to analyze the performance of dynamic memory allocators designed for highly parallel architectures such as the GPU. We build our tests in compliance with the design goals specified in Section 7.1. These benchmarks measure the allocation and free performance of the allocators, data access performance, and compute additional information such as fragmentation and overhead.

Operator performance The first interesting factor is the time it takes to fulfill an alloc or free request. This factor strongly depends on the internal state of the allocator (the allocation history and assigned memory), the number of threads being executed, the number of threads concurrently allocating/freeing data and the size of the requested memory.

To construct a realistic and at the same time challenging scenario, we test the allocators performance across multiple kernel calls. A thread can either allocate memory or free memory that it has allocated before. The probability that it does so is given by the test parameters p_{alloc} and p_{free}. After a couple of kernel calls, the ratio between allocated and free memory will approach a constant. Subsequently to this initial warm-up phase, we measure the average number of cycles required for a single alloc or free. Furthermore, the variance in cycles between different allocation requests is of interest. The coefficient of variation can be seen as a characteristic value of the allocator's homogeneity when it comes to diverging execution paths. To simulate different load characteristics, we increase the number of threads until the device's capacity is reached or the allocator runs out of memory. To capture the influence of the requested memory sizes, they are drawn from an uniform distribution.

Data access performance Besides the time for allocating/freeing memory, the time it takes to access the allocated memory influences the overall performance of a program as well. It strongly depends on the access pattern and cache-utilization. We extend the previously designed test to also measure how long it takes each thread to access the memory it allocated.

Meta Information Not only the raw performance measures give information about an allocator's performance. Internal fragmentation, external fragmentation and memory overhead of the required data structures are also important. Using the previously described test, we can measure these factors to analyze the allocator's performance in different situations according to their definition as given in Section 7.1.

7.6 Results

To compare the performance of *ScatterAlloc* with state-of-the-art CPU allocators on current GPU hardware, we have implemented two list based allocators as described in Section 7.2 using CUDA. Thus, we can also compare their performance directly with the allocator provided with the CUDA toolkit 4.0 and *XMalloc* [48]. Our test-system is equipped with an NVIDIA Quadro 6000 with 6GB of graphics memory, of which we assign 16MB to each of the allocators. The *size list* allocator uses ten bins for speeding up the search for differently sized data blocks. The *mp lists* allocator sorts free memory blocks in ten bins per multiprocessor, which makes for an overall bin-count of 140. For our allocator, we use two randomly selected prime numbers $k_S = 38183$ and $k_{mp} = 17497$ as the respective parameters of our hash-function, a pagesize of 4kB, and a super block size of 8 MB. According to our tests, the performance of *ScatterAlloc* is not sensitive to the choice of k_S and k_{mp}, as long as they are large prime numbers.

Figure 7.3 shows the performance of the alloc and free operators for an increasing number of threads, with $p_{alloc} = p_{free} = 0.75$. As the purpose of this test is to assess the impact of concurrent allocation, we stopped the tests after exceeding the point where the GPU was fully utilized (at about 15000 threads). One can clearly see, that the performance of the list based allocators, *XMalloc* without SIMD optimization, and the CUDA toolkit allocator strongly decreases with an increasing number of threads. *XMalloc* with SIMD optimization can reduce the number of memory request served via the global queues and can avoid a performance decrease until approximately 2000 threads are concurrently allocating memory. The performance of *ScatterAlloc* remains almost constant as the thread count increases. At full utilization of the GPU, memory allocation using *ScatterAlloc* is about 100 times faster than the CUDA toolkit allocator and up to 10 times faster than *XMalloc* with SIMD optimization. Our allocator's performance varies with the requested memory size, due to its page based strategy. If the queues in *XMalloc* become empty, allocation takes a lot longer, which causes the variation in *XMalloc*'s performance. The other allocators show little variance for their allocation time. Comparing the two list based allocators, shows that distributing memory requests of different multiprocessors to different memory-locations yields an increase in performance of about 1000%.

To assess *ScatterAlloc*'s memory consumption characteristics, we compare it to list-based allocators. They work similar to allocators for CPUs, which have been tuned for low memory consumption and fragmentation. Figure 7.4 shows the internal and external fragmentation measured for the list based allocators and *ScatterAlloc* for different distributions of allocation request sizes. We set $p_{alloc} = p_{free} = 0.6$, the mean of the size-distribution to 512 bytes, and varied the extent of the distribution from zero to 896 bytes. The performance of the tested allocators was constant for all distributions and equaled the measurements reported in Figure 7.3. Internal fragmentation was very low for all allocators, indicating that *ScatterAlloc* finds good target candidates most of the time. *ScatterAlloc* as well as the mp lists allocator trade external fragmentation for increased allocation performance. Thus, it is not surprising that the external fragmentation of these allocators is very high. *ScatterAlloc* produces more cache hits than the list based allocators, resulting in faster access. As the latency for the L1 cache is about 18 cycles and about 250 cycles for the L2 cache, the results indicate that all three allocators deliver memory in a cache friendly way.

Figure 7.2 shows the temporal development of *ScatterAlloc* for 16384 threads, two super blocks of 8MB, uniformly distributed allocation requests between 128 and 160 bytes, and with $p_{alloc} = 0.05$ and $p_{free} = 0$. While the first super block fills up, the time needed for finding a free spot increases from 10000 to 25000 cycles. The coefficient of variation (CV) for the allocation time also increases, which indicates that for some threads it becomes more difficult to find a free spot. When there is hardly any space left for allocation, the external fragmentation drops, and the time spent on a single allocation request reaches its maximum. If the allocation request can not be served by the first super block, the next super block is opened up and the allocation time drops below 10000 cycles again. The relative overhead of the allocator for a reasonable fill level is below 1%. As the overhead is constant per super block, the relative overhead decreases with increasing fill-level.

For memory requests larger than the used page size, *ScatterAlloc* also shows a linear performance decrease with increasing number of threads. For a single thread allocating two pages takes approximately 8000 cycles. For every additional thread allocating data, the average performance decreases with approximately 3000 cycles.

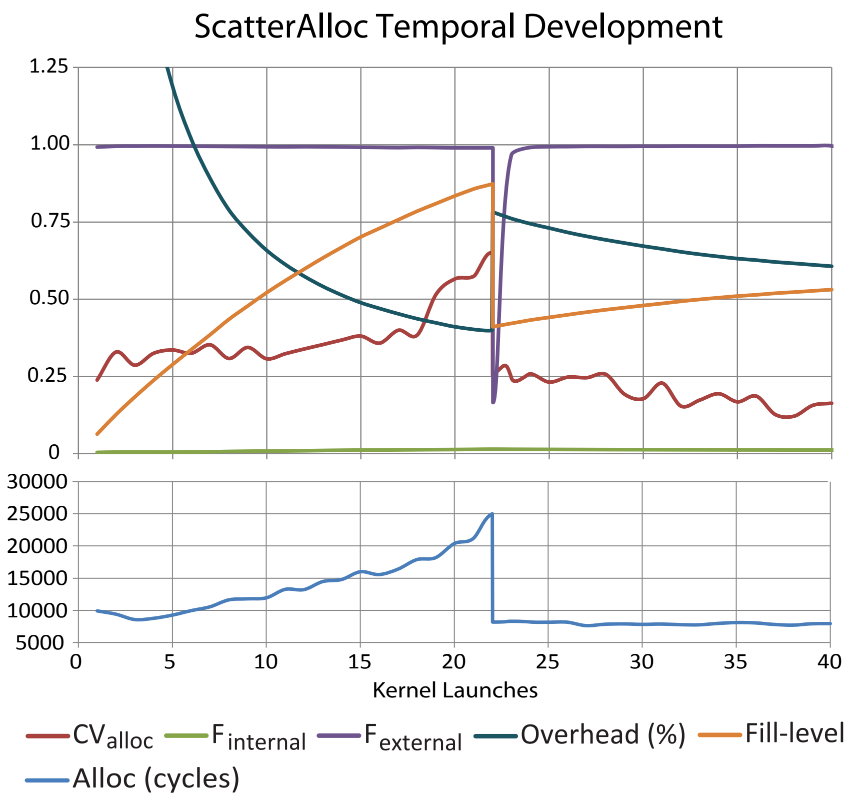


Figure 7.2: Temporal development of *ScatterAlloc*. 16384 threads keep requesting memory chunks between 128 and 160 bytes. As the first super block fills up, the time needed for finding a free chunk starts to vary (CV) and thus increases. After a fill level of about 80% is reached, the next super block is opened up and performance recovers. Internal fragmentation is very low, while the external fragmentation is linked to allocation speed. The relative overhead of *ScatterAlloc* stays below 1% after a reasonable fill level has been reached.

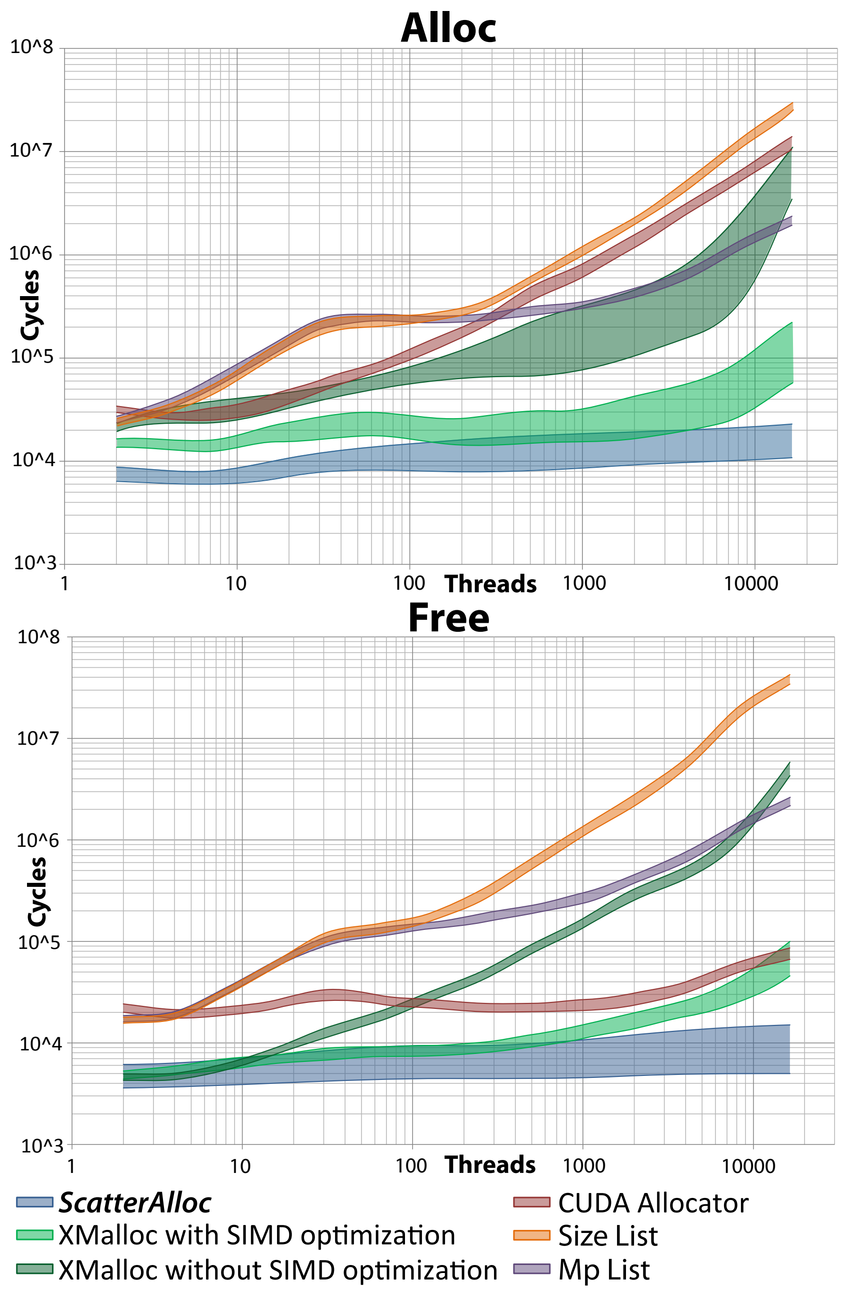


Figure 7.3: Number of clock cycles needed to serve a single alloc or free for 16, 32, 64, and 128 bytes as the number of threads concurrently allocating memory increases. The performance of the list-based allocators, *XMalloc* without SIMD optimization, and the CUDA toolkit allocator strongly decrease with increasing thread count. *XMalloc*'s SIMD optimization can postpone the performance decrease until about 2000 threads are active. Contrary, the performance of *ScatterAlloc* is almost independent of the thread count, outperforming the other allocators over the entire test interval. The CUDA toolkit allocator failed in the test runs with 128 byte allocation size. Thus, this data had to be omitted from above measurements.

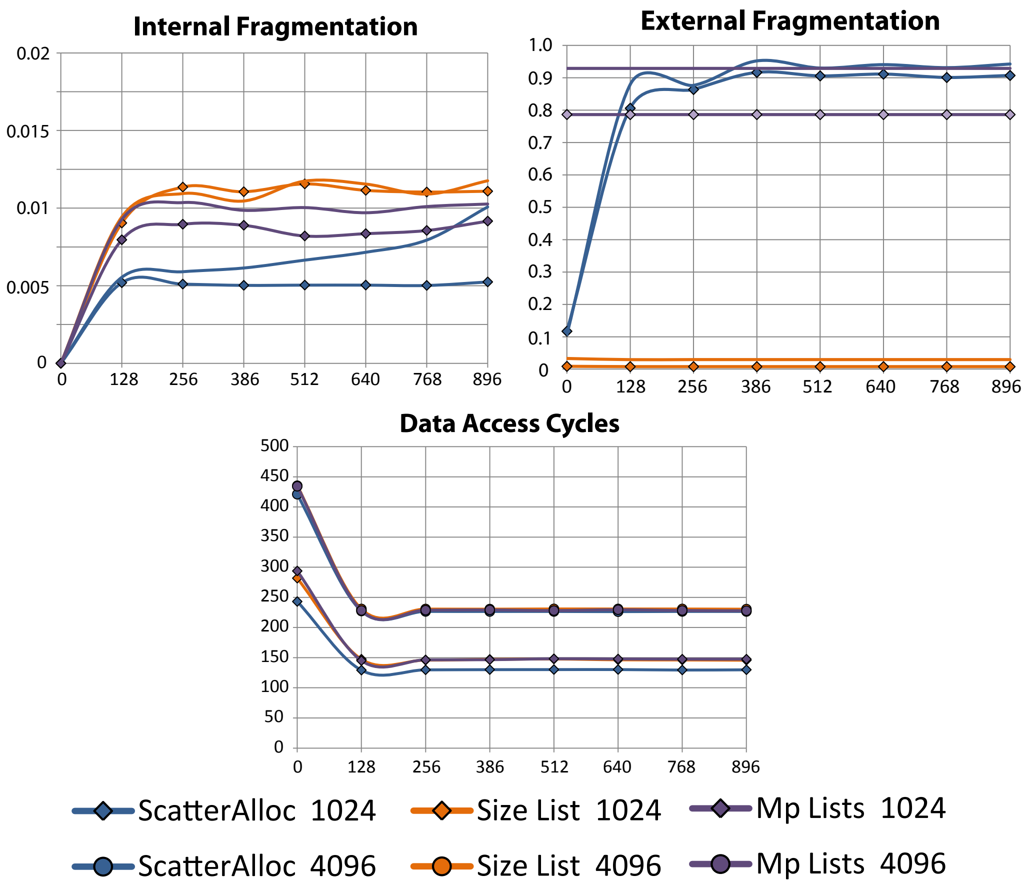


Figure 7.4: Internal and external fragmentation as well as data-access performance for the list-based allocators and *ScatterAlloc* for increasing variation in allocation request size (in bytes), for 1024 and 4096 threads. The mp lists allocator and *ScatterAlloc* both trade fragmentation for speed. Although *ScatterAlloc* uses a page-based memory model, its internal fragmentation is comparable to the list-based allocators. Comparing the achieved access times to the latencies of the L1 cache (18 cycles) and the L2 cache (250 cycles) indicates that all allocators deliver memory in a cache friendly way, whereas *ScatterAlloc* slightly outperforms the other allocators for a lower number of threads.

7.7 Discussion

In this chapter, we described how established CPU-based allocation methods can be tailored to the GPU. However, we also showed that these methods are still too slow to meet the demands of massively parallel programs. Hence, we have implemented our own allocator, *ScatterAlloc*, which turns out to be more reliable and about 100 times faster than the built-in CUDA memory allocation function and up to 10 times faster than the to our work related *XMalloc*.

With *ScatterAlloc*, we have provided a memory allocator that serves memory requests in approximately constant time on the GPU, independent of the number of threads concurrently accessing the allocator. Thus, we can state that **O3** is entirely fulfilled.

Chapter 8

Rendering

Contents

In this section, we show that our methods accelerates selected computer graphics techniques significantly and open new possibilities for algorithmic advancement. For all comparisons between a CUDA and a Softshell implementation, we do not alter the algorithms themselves. We only replace the code needed for issuing kernels by the Softshell interface implementations.

8.1 View-dependent mesh simplification

For our first comparison, we have implemented HDS [80] with a kernel launch behavior similar to state-of-the-art approaches [153]. In a preprocessing step, an octree is built and then traversed during rendering. Each node of this octree can either be expanded or collapsed. Expanded nodes increase the local detail by adding triangles to the mesh, while collapsed nodes define vertex positions.

Our baseline CUDA implementation manages the following three data structures with the help of the highly optimized CUDPP library: a set of active octree nodes, a set of boundary nodes, and a list of active triangles. For each level, two custom kernels (for analyzing and inserting nodes) and two parallel scans (for active nodes and boundary nodes) are called. An additional scan is run to layout vertices in memory before two kernels update the vertex positions and emit triangles.

The Softshell implementation manages all data dynamically during execution. Hence, there is no need for CPU interaction until the mesh is completely constructed, as shown in Figure 8.1. The *traverse event* creates a single work item which maps to the root node of the octree. The entire octree traversal happens in the *traverse procedure.* During the execution of one node, work items for every child node are generated. The work aggregation collects these items and again assigns them to the *traverse procedure.* Each time the traversal reaches a boundary node, a work item is issued for the *generate vertex* procedure. Triangles are generated directly in the *traverse procedure.* The execution order is controlled by Softshell, reducing the necessary programming effort in comparison to the kernel-based implementation. This fact is also captured in the lines of code required for programming the two models: The CUDA implementations has 213 lines of code, while the Softshell implementation consists of 184 lines.

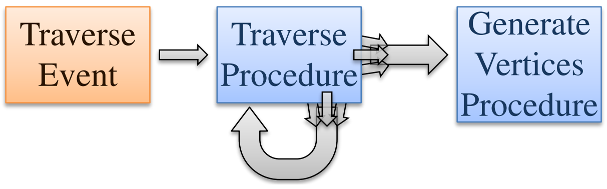


Figure 8.1: Octree-based mesh simplification in Softshell: A single initial work item is generated by the event (orange). The traverse procedure is executed for every octree node, issuing new work items for the traversal. At boundary nodes items for the generate vertices procedure are emitted. Softshell automatically combines individual work items and issues them for coherent execution.

For imbalanced octrees, the Softshell implementation is more than two times faster than the kernel-based implementation, as shown in Table 8.1. This performance gain is mainly due to the ability to leave the control at the GPU and the ability to draw parallelism from the breadth and depth of the octree. Because the *Traverse procedure* creates new work and the *Generate Triangle procedure* does not, the number of available workpackages (and also the utilization) is increased by prioritizing workpackages for the *Traverse procedure.*

8.2 Path Tracing

When implementing a path tracer [53] on the GPU, the greatly varying number of bounces of secondary rays prohibit coherent execution. Using the Softshell processing model, this problem can automatically be addressed by the third-tier scheduler.

Our sample implementation consists of a single procedure that implements a backward monte-carlo path tracer. For each of the 800×800 pixels, a single workpackage is created. Every thread is mapped to one of the 512 rays randomly shot through each pixel. Paths are traced until their contribution falls below 0.1%. Bounces are modeled in a single loop, with the third-tier scheduler being invoked every 10^{th} iteration. As test scene, we used a Cornell box with a varying number of spheres influencing the number of diverging threads.

The results provided in Table 8.2 show that in this special example the use of dynamic thread regrouping can increase performance to a certain extent. Although the probability of a thread to be stalled during an iteration is $50\% - 76\%$ and the execution time for this example is about a minute, the performance gain achieved by thread regrouping is only $4 - 15\%$. This confirms previous findings that it is hardly possible to gain performance using dynamic thread regrouping for interactive applications [105]. Still, this example shows that for long running algorithms with a high rate of thread divergence, dynamic thread regrouping can boost performance.

While the previous example required considerably fewer lines of code for the Softshell implementation, we encounter the reverse situation in this case. The Softshell implementa-

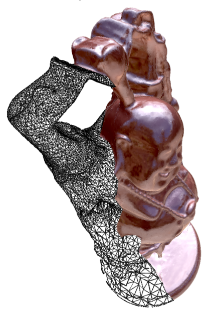
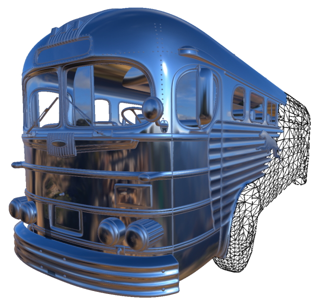
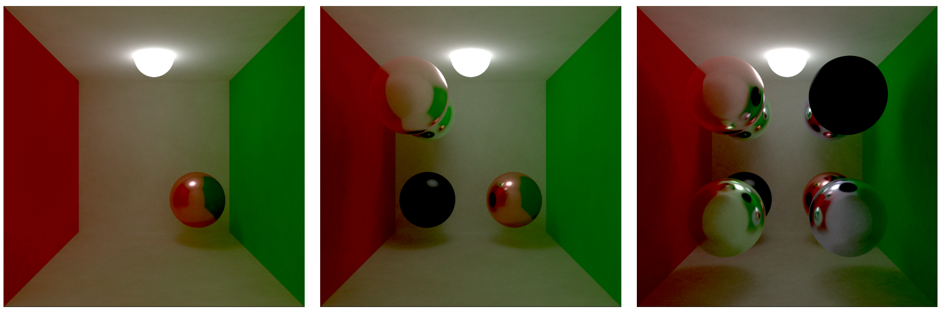
	Happy Buddha			Dragon			Bus		
									
	lev.3	lev.6	lev.9	dist	mid	close	dist	mid	close
vert.	812	56k	796k	3.8k	143k	286k	130k	443k	1.6M
nodes	65	8.3k	362k	501	27k	66k	15k	98k	227k
CUDA	1.77	3.28	9.66	2.75	5.23	7.63	7.19	10.9	36.8
Softshell	0.87	1.66	9.15	0.85	2.48	3.47	3.11	5.42	24.9

Table 8.1: Octree-based mesh simplification: The Happy Buddha dataset is constructed to a fixed octree level. The other two models are constructed based on the current view (from distant to close), for which the decreasing detail in the back leads to inhomogeneous traversal depths. The more nodes are used in the construction, the more vertices (vert) are created and the higher is the data parallelism per level. The construction times for both techniques are given in milliseconds. For balanced octrees with many nodes (Happy Buddha level 9), the kernel-based CUDA implementation is as fast as the Softshell implementation. In all other cases, Softshell is superior because it leads to a better hardware utilization.

tion consists of 95 lines, while the CUDA implementation needs 54 lines. The reason for this change is the fact that this application consists of a single kernel only, which can be expressed very efficiently in CUDA C, while Softshell requires some boilerplate code to be written to individually model all scheduling entities.

This example can be slightly altered to demonstrate the utility of the second tier scheduler. Instead of creating one workpackage for each individual pixel, we create one workpackage for each patch of 4×4 pixels and initially shoot only four rays per pixel into the scene, leading to 64 threads per workpackage. After tracing all 64 paths, we compute the color variance among the paths and resubmit the workpackage, setting its priority proportional to the computed variance. In this way, we prioritize areas for which we are uncertain about the estimated pixel colors. When sending more rays into the scene, we assume that the color converges to its real value and thus reduce the workpackage priority every time the workpackage is resubmitted. In this setup, the second tier scheduler deals with fully dynamic workpackage priorities. Because uncertain image areas are executed first, the rendered image converges to the ground truth faster than a system, which iteratively shoots the same number of rays through all pixels, as shown in Figure 8.2.



div.	0.52	0.63	0.76
CUDA	50.4s	57.4s	66.8s
no T3	52.3s	59.4s	68.9s
ours	50.2s	54.8s	59.6s

Table 8.2: Path tracing with varying number of objects. The more spheres are in the scene, the higher the probability that a thread has to wait for others to finish (div). Using Softshell with no third-tier scheduler (no T3) demonstrates Softshell's overhead for workpackage management when compared to CUDA. As the third-tier scheduler is activated (ours), diverging threads are regrouped, leading to performance increases of up to 15%.

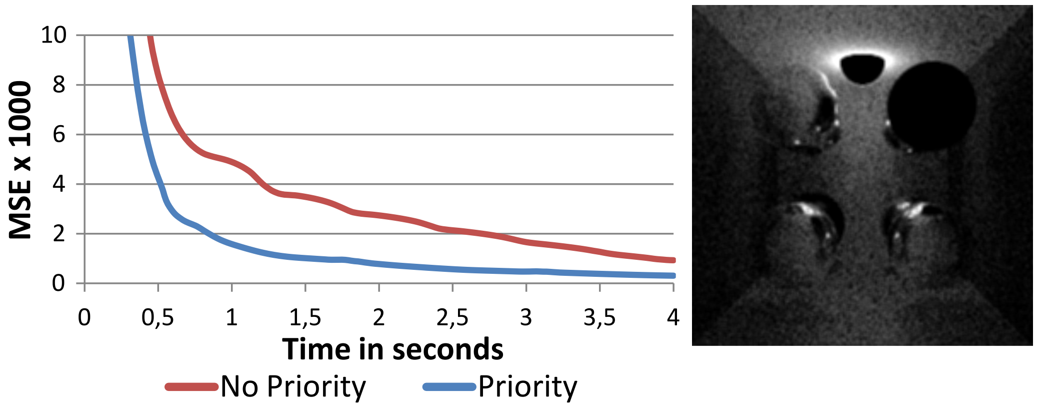


Figure 8.2: Priority-based path tracing in Softshell for test scene three: Adding rays to uncertain image regions first (blue), creates high quality images with lower mean squared error in comparison to the ground truth more quickly than adding rays to pixels uniformly (red). This behavior is implemented in Softshell by dynamically adjusting the priorities of image patches according to the color-variance within the traced paths. The image on the right visualizes the number of traced rays (*cp.* result image in Table 8.2). For white areas, many rays have been shot into the scene, while black areas have been sampled sparsely. Note that uniform regions, like the light source and the black sphere have hardly been sampled.

8.3 Image-based visual hull rendering

The image-based visual hull (IBVH) algorithm computes depth images at novel viewpoints directly from segmented camera images [86], as shown in Figure 8.3. Unlike 3D reconstruction and rendering, it does not require an intermediate data representation, such as a mesh or a voxel grid. It can therefore be classified as an image-based rendering algorithm, and is especially useful for interactive systems that need to produce output images with as little latency to changes in the camera images as possible.

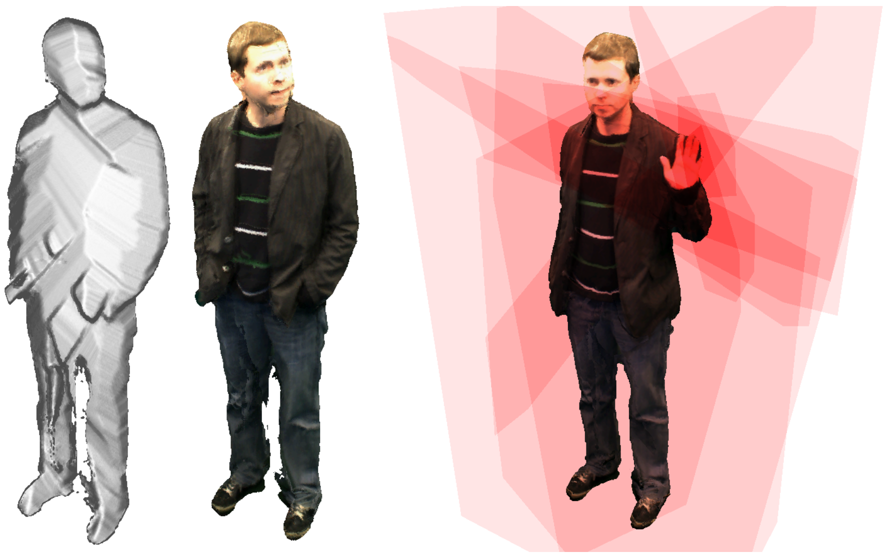


Figure 8.3: A novel viewpoint rendered from 10 camera images by using the image-based visual hull algorithm (left). The redraw buffer encodes the magnitude of change in scene geometry as red intensities, which is directly used as workpackage priority in our Softshell implementation. Images used with permission from Hauswiesner *et al.* [38].

In previous work Hauswiesner *et al.* [38] have shown that computing the IBVH can be restricted to image areas where the underlying geometry has changed without sacrificing image quality. This is achieved by detecting changes in scene geometry and recomputing only image areas where the magnitude of change is above a certain threshold. The geometry of other image parts is reused from the previous frame by means of frame-to-frame image-warping. The threshold directly drives the output quality, but has only indirect effect on the runtime. For interactive systems, however, it is often desirable to directly constrain runtime and maximize the image quality within the available time frame.

Constraining the runtime of a rendering process usually is a very challenging objective. Often, the runtime of rendering kernels can not be estimated before execution, because the scene complexity can vary over time. During GPU execution, the CPU has only limited control over the GPU and can not stop the processing in a well defined manner. After execution, the desired time frame might have been already exceeded. When exploiting scene coherence for IBVH rendering, the area with changed geometry is not the same for every frame. Measuring the amount of changed geometry and inferring the execution time is slow and unstable: the execution time also varies with the complexity of the geometry given as the number of ray-silhouette intersections along each viewing ray. Defining an execution configuration with a certain runtime before execution is therefore not possible. Returning the control flow to the CPU periodically is also not a good option, because it synchronizes the pipeline too often.

This situation highlights the necessity of GPU-based scheduling, as provided in Softshell. Given access to the execution time and control over the execution flow, it is easy to impose constraints on the runtime of the computations. Workpackages are created for the entire image. During the execution of each workpackage we can query the remaining time, and decide whether this block is computed by the IBVH algorithm or reused from the previous frame. Reusing is achieved by image-warping: an operation with fixed complexity whose runtime can be predicted easily. Such a GPU-based scheduling decision is far more fine grained than what can be achieved on the CPU, where usually the decision amounts to whether a grid is launched or not. We also set the priority of workpackages to the average change magnitude of their associated region, so that image areas with substantial changes will be processed first. In this way, we maximize the utility of the given time frame. By obeying the time constraint precisely, the rendering process does not stall or delay other vital system tasks, such as user interaction. In this way, quality can dynamically be traded for execution time to guarantee the frame rate, as shown in Figure 8.4.

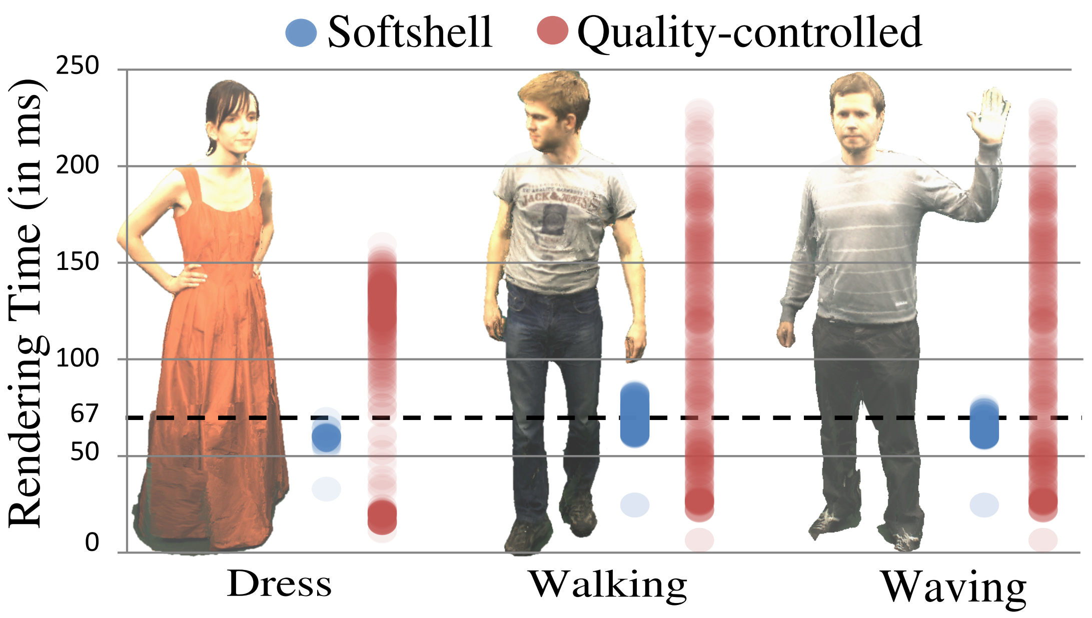


Figure 8.4: Image-based visual hull (IBVH) rendering with a target time-constraint of $67ms$ to match the camera frame rate. We use a low-latency image warping approach for image areas of little change and a full IBVH construction for areas of high change. The algorithms differ in the heuristics used to choose between IBVH and image warping. A fixed threshold determined prior to a kernel-launch does not follow the target frame time (red dots). The Softshell implementation makes this decision dynamically, based on the remaining time. Work of higher-priority is scheduled first, generating the highest possible quality in the specified time (blue dots).

8.4 GPU X3D parser

As the demand for high-quality graphics rises, art assets rapidly grow larger. Parsing a file containing a large model can take a significant amount of time. Using the GPU for parsing model files can speed up this process. However, the X3D file format is composed of many independent constructs, thus, writing efficient parallel GPU code for parsing is no trivial task. Using the BSGP programming model, such a system can be implemented with less programming effort [46]. For the execution, still about 80 kernel-functions are automatically created.

For comparison, we provide our own X3D parser in Softshell, which works similar to the BSGP X3D parser. We divide the parsing into two events: At first, we use six procedures (two of them implement sorting and the scan algorithm) to generate a skeleton of the XML-tree. The scan and sort algorithms are called multiple times, for data sizes that individually could not fully occupy the GPU. Softshell schedules these algorithms concurrently whenever data or parts of the data become available. Due to its kernel-based structure, the BSGP parser executes these steps sequentially, losing performance in comparison to our implementation.

Figure 8.5: GPU-based X3D parsing: The 13MB test scene contains 2300 nodes of which 1400 are shape defining. Using the BSGP [46] implementation, parsing takes $71.25ms$. Our Softshell implementation parses the scene in $36.56ms$.

In a second step, we parse the XML-tree using one procedure per supported X3D-tag. In this way, the parser can easily be extended to support new X3D-tags. The execution starts at the root node and a new workpackage is created for each encountered node. Thus, groups of threads work on the individual nodes in parallel, while the thread count is dynamically adjusted to the node type. Depending on the node type, new workpackages for child nodes are generated and/or output data is written either to the vertex buffer or a state buffer, which is then used during rendering to adjust the OpenGL state machine. Softshell schedules workpackages for the different X3D-tag procedures concurrently and therefore increases the GPU utilization. In this way, our X3D parser can launch a higher number of coherently executing thread groups than the BSGP parser. Additionally, the control flow remains entirely on the GPU until the model is ready to be rendered, and a costly back and forth between GPU and CPU is avoided. For a complex scene (Figure 8.5), parsing takes BSGP 71.25ms, while the Softshell implementation is nearly twice as fast with 36.56ms.

8.5 Real-time Reyes Pipeline

The Reyes rendering pipeline [19] is primarily used in cinematic productions, as it efficiently synthesis images of high quality. However, an implementation on graphics processors is a challenging task, as the pipeline is recursive, irregular and has unbounded memory requirements. A typical Reyes pipeline consists of the following six stages:

1. **Bound** In the bound stage, the input primitives are assembled. In a first step, each primitive's bounding box is intersected with the view frustum to cull primitives which are not visible. Additionally, primitives which completely face away from the camera can be removed.

2. **Split** In the split stage, primitives are recursively split into smaller patches until their bounding box indicates that they are small enough for dicing. Before issuing primitives for another split, they can be tested against the viewing frustum to remove invisible sub-patches.

3. **Dice** In the dice stage, every patch is subdivided into micropolygons according to a regular grid. The grid size is normally chosen to generate at least four micropolygons for each screen pixel.

4. **Shade** In the shade stage, every micropolygon vertex is shaded using a custom shader. The micropolygon color is later determined by interpolating the vertex colors.

5. **Sample** In the sample stage, every pixel is subdivided into a fixed number of subpixels. For every subpixel, a sample positions is chosen using a random displacement from the subpixel center. For every sample location covered by a micropolygon, the micropolygon's color and depth is determined and stored.

6. **Composite** The final stage of the pipeline is the compositing step, which determines the color of every pixel combining its subpixel colors.

Previous attempts to bring this pipeline to the GPU always split the pipeline into multiple kernel launches: Patney *et al.* [106] use a four-stage model. The first stage contains the recursion and is split into multiple kernel launches. Between these launches the buffers holding the input data for the next launch are compacted. Zhou *et al.* [154] use an eight-stages model, whereas three stages are only concerned with making scheduling decisions. Overall many more kernel launches are needed. Their system is not real-time, but very general and, thus, can produce production quality images. Tzeng *et al.* [141] propose a five-stage model, whereas the first stage uses persistent threads to handle the recursion in a single kernel call. The remaining stages are implemented as individual kernel launches.

To form a challenging task for our scheduling framework, we model the entire pipeline as a single algorithm leading to a kernel launch in our model. To make the task even more

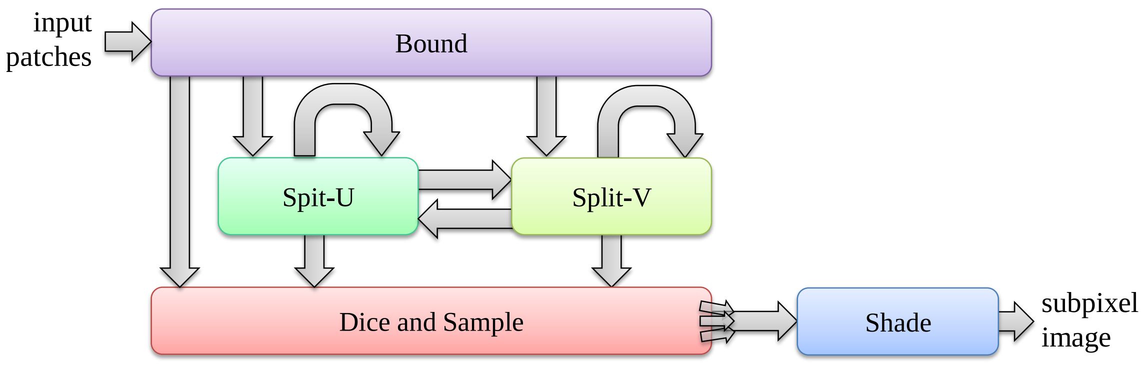


Figure 8.6: Our Softshell Reyes pipeline consists of five procedures. Bound operates on item sets with 16 threads, both split procedures expect item sets for four threads, dice and sample operate on workpackages with 256 threads, and shade operates on items. Shared memory requirements reach from 0 to 48 bytes per thread and register usage varies between 30 and 63, resulting in an algorithm with very diverse characteristics.

challenging, we exchange the order of the shade and sample stage, introducing an exact per-sample lighting evaluation. Our pipeline is shown in Figure 8.6. As input data we use Beźier patches with a 4×4 control mesh. The bound stage assembles the input data and computes the position of every control vertex relative to the view frustum to perform frustum culling. To check if the entire patch faces away from the camera, each thread transforms its vertex to screen space and writes the vertex position to shared memory. Then, the orientation of each face is computed and if all faces face away from the camera, the patch is culled. We require one thread for each control vertex, resulting in a 16 threads item set procedure with 128 bytes of shared memory to communicate the vertex positions. We divide the split stage into two procedures. Each procedure splits a patch along one dimension. To perform the split, we use one thread to operate on one row or one column of the input patch, using four threads per procedure. Using four threads requires an item set procedure. Additionally, this procedure uses 192 bytes of shared memory to share the patch data between all threads. The dice procedure dices each patch 15 times along each dimension, generating 16×16 vertices. In the first part of the procedure, we use one thread per vertex. In the second part, we use one thread per micropolygon to perform sampling. This setup requires synchronization between the threads and thus results in a workpackage procedure using 256 threads. To store the data for all mircopolygons in shared memory, 3072 bytes are required. The shade stage uses one thread per sample location to evaluate the lighting model for each subpixel. No cooperation between threads is required, yielding an item procedure with no shared memory requirements. The number of registered used by the individual procedures also varies: The bound, split, dice, and shade procedures use 38, 63, 40, and 30 registers respectively. Overall the pipeline shows strongly varying procedure characteristics and also requires a significant amount of shared memory. To composite the subpixel colors and display the result we use OpenGL.

To evaluate our Reyes pipeline, we compare seven different implementations. The first two use the kernel with queues implementation. KwQ copies the input data to

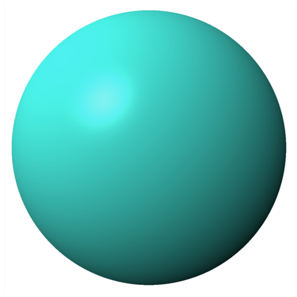
Table 8.3: Reyes test scene statistics: number of input primitives, calls to the split procedures, generated micropolygons, and final shaded subpixels.

	Primitives	Split U	Split V	Micropolygons	Shade
Sphere	8	19k	13k	8.0M	0.26M
	8	74k	51k	32.1M	1.06M
	8	297k	205k	128.5M	4.23M
Teapot	32	20k	19k	10.2M	0.28M
	32	82k	77k	40.8M	1.11M
	32	328k	310k	163.3M	4.45M
Killeroo	11500	12k	14k	9.4M	0.24M
	11530	61k	67k	99.0M	0.96M
	11532	269k	271k	140.8M	3.84M

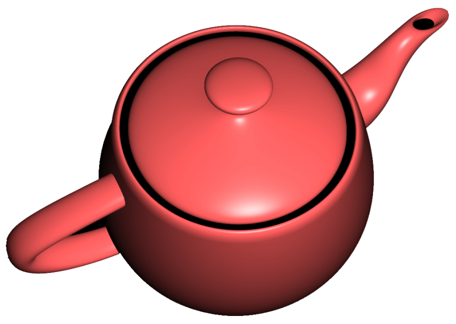
shared memory, whereas KwQ_A accesses the data in the queues directly. A basic dynamic parallelism implementation (DP) launches new kernels directly during the execution of procedures, whenever new data is generated. HDP and HDP_A correspond to our hybrid dynamic parallelism with controller implementation with and without access modifiers. The details of all approaches are discussed in Section 6. $Softshell$ corresponds to the full featured Softshell implementation with a versatile megakernel. $Softshell_{loc}$ uses locally buffered queuing for the shade procedure to avoid costly global memory access for every micropolygon. As test scenes we used a simple sphere, the Teapot, and the Killeroo dataset, as shown in Figure 8.7. The screen resolution was set to 1920×1080 pixels with subpixel counts of 1, 4, and 16. These settings influence the number of execution steps and intermediate data, as shown in Table 8.3.

The results of the Reyes tests are shown in Table 8.4 and Table 8.5. On the GTX 680 with a screen resolution of 1920×1080 and a single sample per pixel, real-time frame rates can be achieved, if Softshell is used with locally buffered queuing. Both kernel-based implementations are slower than the Softshell implementation with global queuing. $Softshell_{loc}$ is on average three times faster than $Softshell$, four times faster than KwQ_A and five times faster than KwQ. Dynamic parallelism is not supported on the GTX 680. For 1920×1080 pixels with 16 samples per pixel, we still achieve about four frames per seconds for all datasets. Considering that more than 100 million micropolygons are generated from only a small number of input primitives, these results can still be considered as a success.

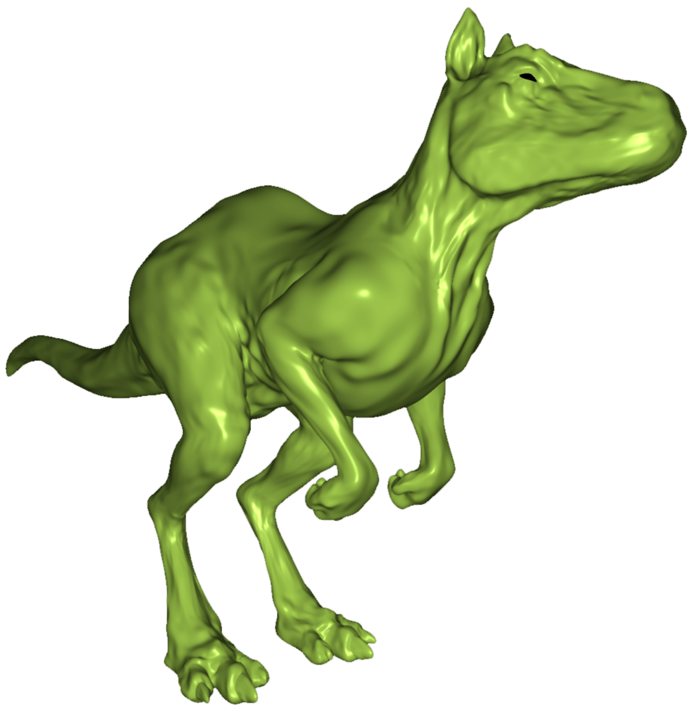
On the GTX TITAN, the impact of queues in shared memory is again smaller than on the GTX 680. Also, KwQ_A, HDP_A, and $Softshell_{loc}$ generate similar frame rates. Again, the overall performance was similar for all data sets. While $Softshell_{loc}$ achieved the best performance for one sample per pixel, KwQ_A achieved the best performance for 16 samples per pixel. The basic dynamic parallelism implementation was very slow in comparison to all other methods, taking seconds to generate a single frame. With the best methods, a 1920×1080 image without super sampling can be generated with up to 200 frames per second on the GTX TITAN. If four samples are taken per pixel, the frame rate



(a) Sphere, 8 input primitives
128M micropolygons



(b) Teapot, 32 input primitives
163M micropolygons



(c) Killeroo, 11532 input primitives
140M micropolygons

Figure 8.7: The Reyes test scenes used during evaluation.

Table 8.4: Softshell Reyes Rendering on a GTX 680 with 1920×1080 viewport and different subpixel counts: rendering times in ms.

	Subpixels	KwQ	KwQ_A	$Softshell$	$Softshell_{loc}$
Sphere	1	88.5	75.3	52.6	16.4
	4	342.7	293.9	206.1	61.3
	16	1350.9	1160.1	820.8	240.8
Teapot	1	106.7	90.7	61.3	20.3
	4	415.6	350.8	240.3	78.7
	16	1640.7	1390.7	960.9	310.2
Killeroo	1	105.2	87.8	59.7	21.7
	4	390.1	330.9	225.5	74.2
	16	1520.8	1299.8	880.3	280.1

Table 8.5: Softshell Reyes Rendering on a GTX TITAN with 1920×1080 viewport and different subpixel counts: rendering times in ms.

	Subpixels	KwQ	KwQ_A	DP	HDP	HDP_A	$Softshell$	$Softshell_{loc}$
Sphere	1	11.2	7.2	6482	11.4	6.7	8.3	5.3
	4	35.7	21.0	26690	41.8	24.1	31.8	20.3
	16	134.1	72.2	-	135.3	77.3	126.2	80.4
Teapot	1	13.8	8.0	7278	14.2	7.0	10.2	6.7
	4	43.6	24.4	29840	44.7	27.3	40.2	26.0
	16	163.2	99.6	-	165.3	102.8	160.1	103.5
Killeroo	1	11.8	7.1	6860	12.3	6.9	10.1	6.6
	4	40.1	22.1	28312	43.1	23.7	37.9	24.6
	16	153.4	86.6	-	157.9	89.6	159.8	99.5

is reduced to approximately 50 frames per second. A 16 times sub-sampled Reyes image can be rendered with 10 frames per second.

To analyze the behavior of the different execution strategies in more detail, we record the exact execution time and duration of each procedure. These records are shown in Figure 8.8, 8.9, 8.10, and 8.11 for the teapot data set and an image resolution of 400×400 pixels. Every individual procedure execution corresponds to one rectangle. The darker part of each rectangle corresponds to the time needed for enqueue. The block width relates to the number of threads active in the procedure. KwQ shows a relatively slow start and the delay for reading back the queue fill-rates to the CPU is clearly visible. However, when there is sufficient input data for the dice procedure available, the utilization is very good. Up to five dice procedures are executed on one multiprocessor concurrently. On the GTX TITAN, starting procedures into different streams also leads to concurrent execution on the device, especially increasing utilization during the execution of the split procedures.

For the split procedure, enqueue makes up about one quarter to one third of the execution time. Many instances of the dice procedure do not generate samples. If they do, enqueue consumes approximately one fourth of the execution time. HDP shows a less coherent behavior. Especially in the beginning the utilization is very low, when many procedures with only few threads are being executed. When there is overall sufficient data available, the utilization is increased and up to six dice procedures are executed concurrently. Still, it seams that dynamic parallelism sometimes runs into situation where it is unable to launch different kernels, which results in reduced utilization. Softshell with our versatile megakernel quickly increases the utilization in the beginning of the execution and hardly shows any wholes during execution. Also, any combination of procedures can be executed concurrently. The major drawback of the megakernel approach is also clearly visible: only four procedures can be executed concurrently, because the split procedures increase the register requirements for the megakernel to 63. The relative enqueue overhead still is the same as in KwQ. If locally buffered queuing is used, the scheduling characteristics remain similar, however, the execution time reduces by approximately 10% on the GTX TITAN. Taking a closer look at the relative enqueue overhead shows that the time spent on enqueue in the dice procedures is reduced to about one tenth. This fact again shows that queues in shared memory are more efficient than global queues and can result in significant speed-ups if used for the right procedures.

Overall, the performance of our megakernel approach is very competitive, even for strongly varying procedure characteristics with challenging shared memory requirements and millions of intermediate items being generated. Although each kernel call can be individually optimized resulting in lower register counts and lower shared memory requirements than our versatile megakernel, our approach is still on par with our most efficient kernel-based implementations on the GTX TITAN and even four times faster on the GTX 680.

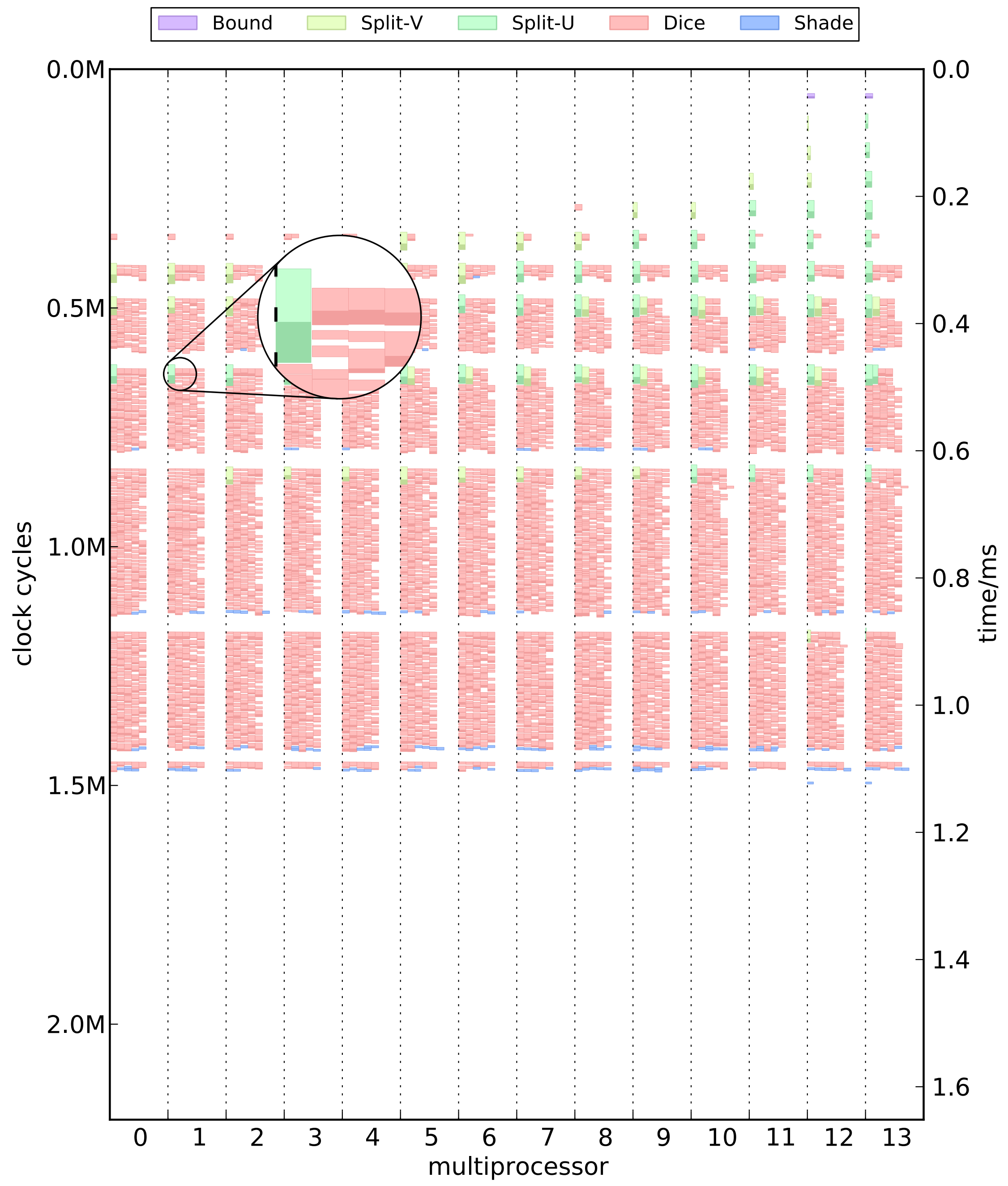


Figure 8.8: Scheduling behavior of the kernels with queues approach on the GTX TITAN for the teapot data set with 400×400 image resolution.

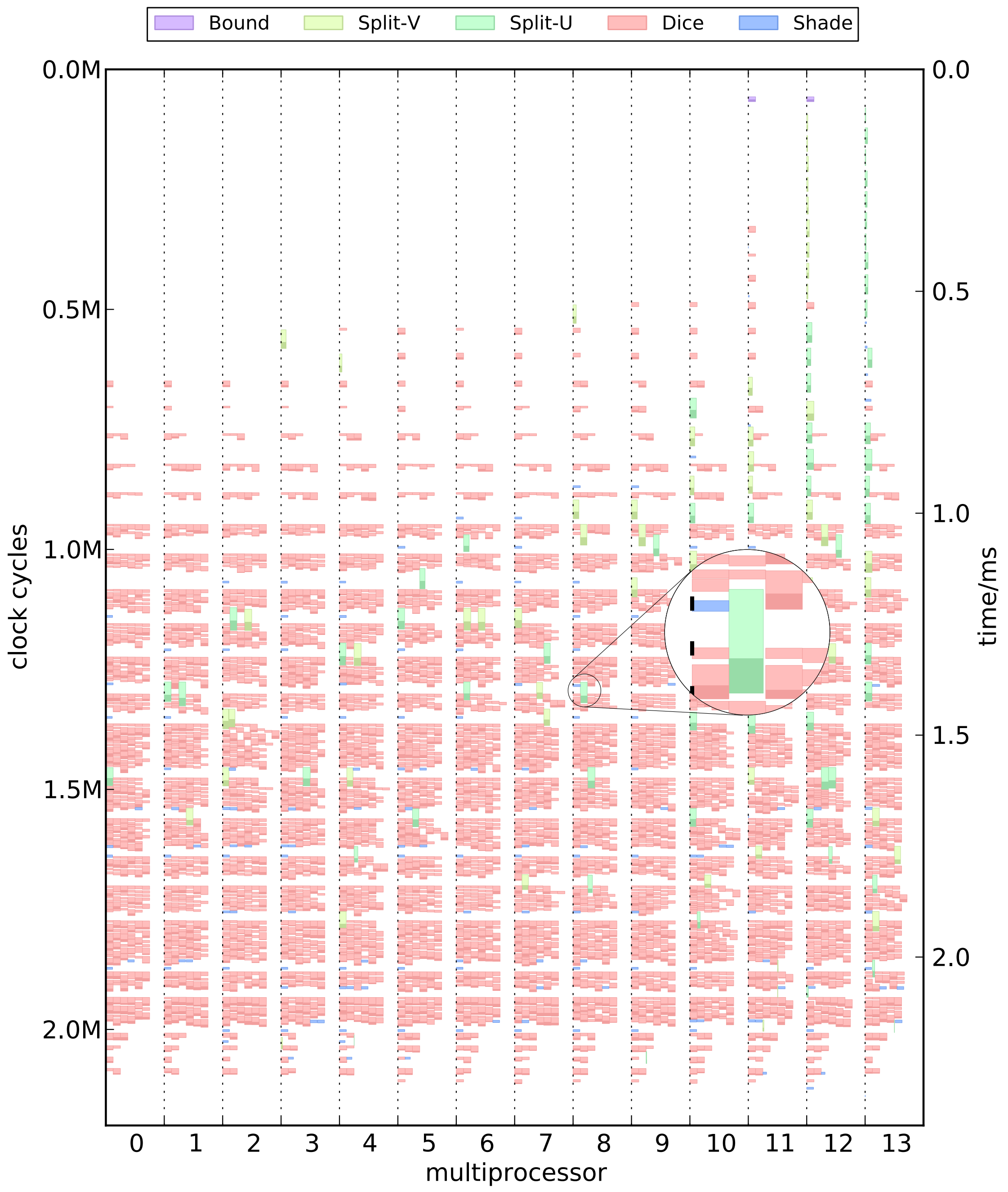


Figure 8.9: Scheduling behavior of the hybrid dynamic parallelism with controller approach on the GTX TITAN for the teapot data set with 400×400 image resolution.

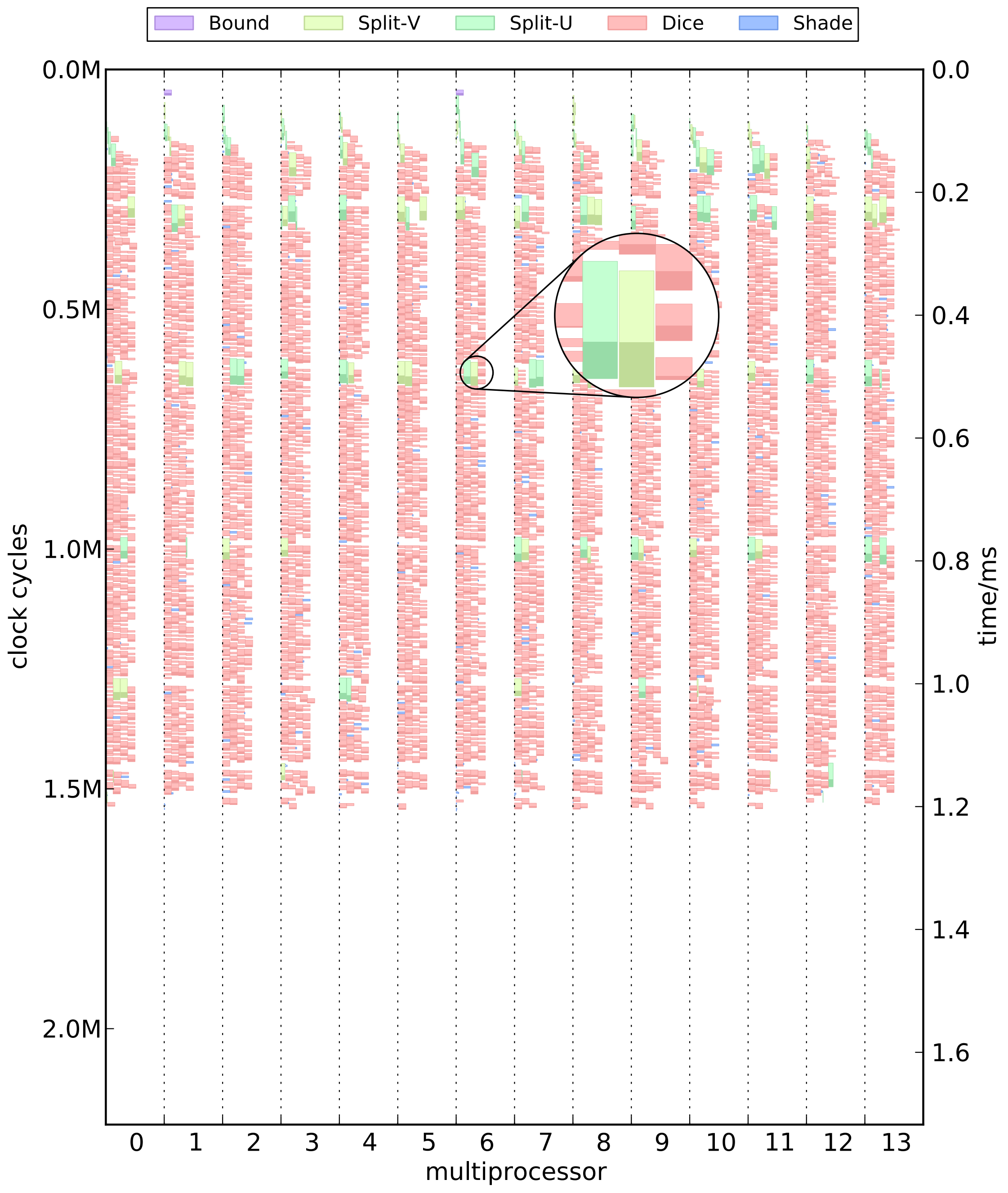


Figure 8.10: Scheduling behavior of Softshell using a versatile megakernel on the GTX TITAN for the teapot data set with 400×400 image resolution.

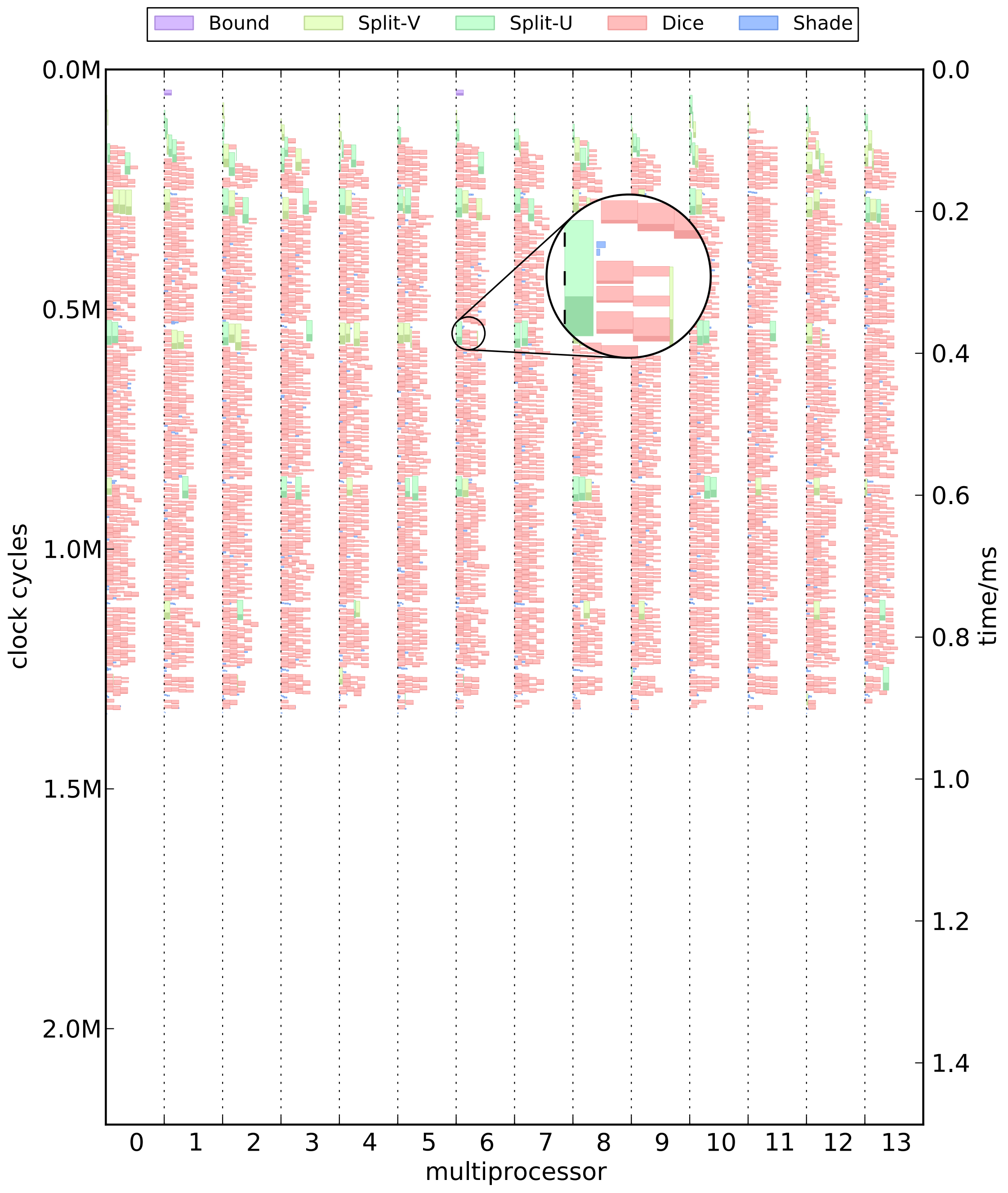


Figure 8.11: Scheduling behavior of Softshell using a versatile megakernel with locally buffered queuing on the GTX TITAN for the teapot data set with 400×400 image resolution.

8.6 Discussion

In this chapter we have shown that it is possible to efficiently map dynamic algorithms with a relatively low degree of local parallelism for the execution on massively parallel architectures. Our tests demonstrate that the feature set of Softshell can improve the performance of computer graphics applications in particular. Our Softshell-based implementations of dynamic mesh simplification proved to be two times faster than the state-of-the-art execution strategies based on the kernel-based execution model (**O1**). We were able to speed up path tracing by up to 15% reducing the divergence between rays (**O5**). Additionally, we could show that the convergence rate of Monte-Carlo-based global illumination methods can be increased when dynamically assigning processing power to those regions that contribute the most to image quality (**O4**). For image-based rendering, we could show that time-aware scheduling can dynamically adapt an algorithm to stay within the tight time budged imposed by camera update rates (**O4**). We also compared Softshell to the BSGP programming model, showing that parsing of model files can be sped up by a factor of two by executing different procedures concurrently and avoiding back-and-forth between CPU and GPU (**O1**). Finally, we presented a real-time Reyes pipeline, providing update rates of 40 frames per second and above for screen resolutions of 1920×1080 pixels with four times super sampling. In this way, our pipeline handles up to 100 million micropolygons in every frame. On a GTX 680 our versatile megakernel is four times faster than the most-efficient kernel-based implementation (**O2**).

Chapter 9

Visualization

Contents

Interactive rendering of volumetric data is important in many fields, such as medicine, geology, and engineering. Virtually all interactive state-of-the-art volume rendering techniques build on the computational power of graphics processing units. Multiple factors can influence the performance of volume rendering algorithms executed on the GPU. For volume rendering, data access times often form a bottleneck, as many samples need to be drawn from the volume. Another performance influencing factor is thread divergence. If not all cores of a SIMD unit can be supplied with the same instruction, a subset of the cores will be disabled. This situation can happen, if threads choose different execution paths or finish early. In a worst case scenario, all threads choose different execution paths for which their execution is completely serialized. If a rendering algorithm needs many passes/kernel calls, switching between GPU and CPU might form a serious bottleneck. The overall GPU utilization is important. If there is not enough parallelism in the algorithm to keep all GPU cores busy, performance will also be affected.

We hypothesize that a combination of these issues might often prevent volume rendering algorithms from utilizing the power of the GPU to a full extent. In this chapter we study the behavior of three different volume rendering implementations and analyze under which circumstances the aforementioned problems arise. Our first case study is direct volume rendering (DVR) [74], which is an image-order volume rendering technique. In DVR, rays are traced through the volume and samples are accumulated along the ray. Our observations can be generalized for other techniques which sequentially sample along rays. As second case, we studied a more complex state-of-the-art illumination model: Image Plane Sweep Volume Illumination [136]. Finally, we looked at particle-based volume rendering (PBVR) [117], as an example for object-order volume rendering.

For each example, we analyze possible problem cases and show where dynamic scheduling strategies can help to overcome these issues and increase performance. For simplicity, we will stick to terms used in stream-processing languages like CUDA, for which a kernel launch roughly corresponds to a rendering pass in a shader-based system. The execution carried out by one thread equals the steps executed for a single fragment in a fragment shader.

Direct volume rendering is probably the most widespread method for displaying dense 3D data. Since the early work by Levoy [74] the raycasting algorithm has been extended in different respects. Many researches focused on improving the performance of DVR, including the skipping of empty space [64, 114] or terminating rays as early as possible [64]. Rapid growth of the computational power of graphics processors and their architectural changes led to a series of advancements in the field, including the use of shaders [130] or CUDA [82] and the interactive rendering of multiple volumes and semi-transparent polygonal meshes [52]. Similarly, other SIMD architectures were employed for high-performance volume rendering, such as the Cell Broadband Engine [57].

Another focus of research in volume rendering is improving the perceptual effectiveness of the resulting images. In particular, improving depth perception using global illumination effects, such as shadows. Several recent studies have indeed shown that depth and relative size perception improves if advanced illumination models are employed [77, 115, 128]. Therefore, many methods which simulate advanced illumination effects have been introduced, such as Monte-Carlo integration [63, 113], deep shadow maps [34], or volumetric scattering and shadowing [115]. A more recent approach proposed by Sunden and colleagues, uses an image-plane sweeping algorithm to perform scattering and shadowing in volumes [136]. This algorithm does not require any preprocessing or stored illumination volumes, which makes it usable for rendering of very large datasets. However, the possibilities for parallel execution of raycasting are limited to one image column, reducing performance. In this chapter we analyze how the performance of this algorithm can be improved using our scheduling approach.

Raycasting is usually performed for datasets represented on structured grids. For unstructured grid data, object order volume rendering techniques are often used, such as projected tetrahedra [87, 127] or splatting [95, 120, 145]. Unlike splatting-based approaches, particle-based volume rendering [21, 117], employs opaque particles and, thus, does not require visibility sorting before projecting onto the screen. This allows easier parallelization in comparison to the aforementioned object order approaches. We explore how the performance of such particle-based approaches may be further improved using dynamic scheduling strategies on the GPU.

9.1 Test System and Datasets

Again, we build our scheduling implementations in Softshell. The baseline implementations are written in CUDA. For all tests, the computations carried out for generating the output image are the same in the baseline implementation and our scheduled version, only the assignment to execution units differs. As a test system, we used an Intel Core i7-940 Quad Core CPU (2.93 GHz) with 8GB RAM and a NVIDIA Quadro 6000 graphics card. For our evaluations, we used three regular-grid datasets with different properties. The Stanford Dragon 64^3 (Figure 9.1) has smooth volumetric structures, the Bonsai dataset $512 \times 512 \times 182$ (Figure 9.4a) has many small structures, and the MECANIX volume $256 \times 256 \times 372$ (Figure 9.4b) combines tissues with different densities. The two unstructured grid datasets we use are a tetrahedralized version of the Stanford Dragon with 1.25 million equally sized cells and the simulation data of a radio frequency ablation (RFA) with 1.25

million cells. Contrary to the Dragon dataset, the cell sizes of this dataset vary by a factor of over 1000.

9.2 Direct Volume Rendering

DVR is one of the best studied volume rendering algorithms. In DVR a ray is sent into a scene for each pixel on screen, and equidistant samples are taken while passing through the volume. The dataset is most often represented as a regular, three dimensional voxel grid of density values. Interpolation methods are used to determine the exact density value for each sample. These density values are then used in a so called transfer function lookup to determine a color and an opacity value which are then used to compose the final color. Using a front-to-back sampling order, sampling can be stopped upon reaching a sufficiently high composite opacity, *i.e.*, if successive samples would not significantly change the pixel color anymore. This strategy is called early ray termination.

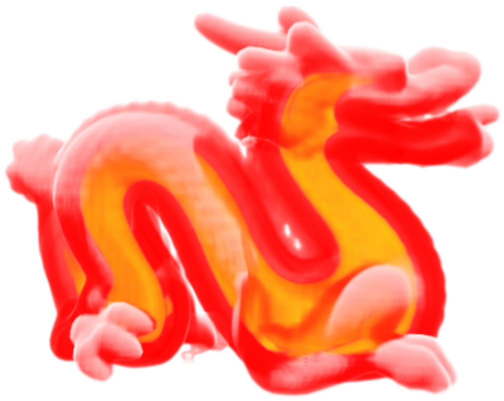


Figure 9.1: The Stanford Dragon, used in the DVR and PBVR case study, is a data set which is simple both in terms of resolution (64^3 voxels) as well as internal structure. We use this dataset in conjunction with a simple transfer function for worst-case tests.

9.2.1 Thread divergence during sampling

One possible problem with early-ray termination and other volume rendering methods which require an unknown number of samples along rays is thread divergence. As warps are executed in lock-step, divergent execution paths within warps can form a problem. If the number of samples for rays within a warp significantly differs, only a fraction of threads are active during the final traversal steps, which reduces the performance.

Intuitive parallelization of DVR utilizes one thread per ray (*i.e.*, per pixel). Most approaches assume that if rays within one warp are close in screen space, the accumulation process along these rays will be highly correlated, leading to little divergence. However, due to data inhomogeneities and varying opacities, threads might not show a high correlation.

We have analyzed the thread divergence while rendering three sample datasets. The dragon datasets represents a very homogeneous example, with only two opacity values defining the transfer function. As second example, we tested the Mecanix dataset with a transfer function setup mainly showing opaque bones, which might lead to thread divergence at the boundaries. The third dataset is Bonsai, which has many small leaves, possibly leading to divergence.

As shown in Table 9.1, the thread divergence for standard DVR (STD) is not very high, with an average warp utilization between 78% and 84% when rendering images of 1024^2 pixels. If we decrease the viewport size, the divergence rises to about 60%, as neighboring rays hit more different structures. The reason for this relatively good utilization might also be found in the fact that we did not use tightly fitting bounding boxes, and thus a large amount of empty space is traversed in a coherent manner. However, DVR in this setup runs at about 60% to 80% peak performance.

9.2.2 Ray re-convergence

Even though locally threads diverge, globally many threads are executing the same code – sampling along the ray. Using Softshell, we apply a thread re-convergence strategy. In the sampling loop, the number of active threads is checked every L iterations. If the number of active threads drops below P%, Softshell interrupts the execution and saves the context of each thread. It then automatically divides the threads into still sampling threads and finished threads. For each type, a list of threads is kept in global memory. When a sufficient number of threads (enough to fill up an execution unit) have been stored in either of the lists, they are submitted for execution. When the execution is restarted, the thread states are recovered and each threads continues its execution where it has been stopped before, with the difference that thread divergence has been removed. However, during the next iterations threads may diverge again. Thus, we use the same mechanisms again.

Using the parameters L and P, we can control how aggressive our re-convergence strategy should be applied, see Figure 9.2. Choosing a low L and a high P leads to stronger divergence reduction. Even though, we could reduce thread divergence with our strategies, this does not necessarily increase performance, because of the overhead of re-convergence (see Table 9.1). This includes the costs of storing thread states, combining the paused threads and restarting them, as well as possible worse cache utilization, as non-neighboring rays might be grouped. A good parameter setting for L and P depends on the used data set and complexity of the operations carried out for each sample. Note that we were not able to completely remove divergence, because we do not break thread groups apart when merging paused threads. For example, if only three threads are missing to form a perfectly fitting group and the next paused thread group has more than three threads, we issue the group which misses three threads for execution. Nevertheless, especially if the utilization is low (below 70%), our re-convergence helped to increase performance, leading to speed-ups of up to 1.54 times. As DVR does not show a high amount of divergence, we would expect higher speedups for more complex volume tracing strategies, such as Monte-Carlo ray tracing with multiple scattering.

9.3 Image Plane Sweep Volume Illumination

Image Plane Sweep Volume Illumination [136] renders volumetric datasets in an image-based scan-line fashion with interleaved scattering and shadowing operations. The authors

Table 9.1: Rendering times (ms) for DVR with different re-convergence strategies. STD is the baseline CUDA implementation, RCVG corresponds to our re-convergence strategy with $L = 20$ and $P = 0.7$, and A-RCVG is aggressive re-convergence with $L = 1$ and $P = 0.95$. For big viewports, groups of rays are more coherent, leading to less divergence and a higher warp utilization (util). If the utilization is higher, fewer loop iterations (its) are overall executed. Due to the overhead of re-convergence a higher utilization does not necessarily lead to a speedup (↑). How well the re-convergence strategy works, strongly depends on the dataset and the already present thread divergence.

		Dragon				Mecanix				Bonsai			
method	image	time	speed↑	its	util	time	↑	its	util	time	↑	its	util
STD	1024^2	46.4	1.00	365k	84%	646.2	1.00	7.02M	78%	694.7	1.00	2.67M	82%
RCVG	1024^2	57.3	0.81	352k	87%	847.8	0.76	6.17M	88%	606.4	1.15	2.53M	87%
A-RCVG	1024^2	69.7	0.67	328k	94%	1199.4	0.54	6.09M	89%	732.8	0.95	2.44M	90%
STD	512^2	28.2	1.00	119k	78%	348.5	1.00	2.35M	72%	451.6	1.00	918k	74%
RCVG	512^2	27.5	1.03	116k	85%	366.3	0.95	2.00M	85%	329.1	1.37	848k	81%
A-RCVG	512^2	31.7	0.89	109k	90%	449.7	0.77	1.92M	89%	357.9	1.26	798k	86%
STD	256^2	16.2	1.00	36.3k	65%	155	1.00	649k	65%	213.9	1.00	277k	61%
RCVG	256^2	12.9	1.26	36.1k	73%	140.6	1.10	532k	80%	147.1	1.45	233k	74%
A-RCVG	256^2	12.6	1.29	31.2k	84%	156.8	0.99	488k	88%	139.3	1.54	216k	80%

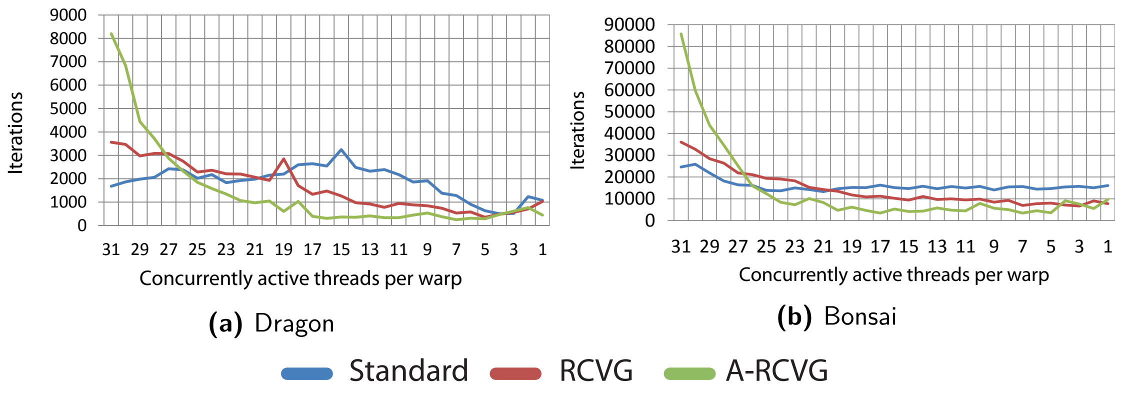


Figure 9.2: Warp utilization for DVR (excluding iterations of fully utilized warps). Using our re-convergence strategy, the number of warps with low utilization is decreased, while warps of higher utilization are generated. While a more aggressive setup clearly generates more utilization, the overhead associated with re-convergence might actually decrease performance.

focus on a memory-efficient method for constructing a deep shadow map concurrently with ray casting. Essentially, the shadow map is created slice-by-slice, while tracing the rays which intersect with the current slice. Because only the current slice is kept in memory, only one slice of rays can be traced at a time. To resolve slice to slice dependency , synchronization via the CPU is necessary after each slice, leading to a multi-pass algorithm, as outlined in Figure 9.3(a). This control switching between CPU and GPU not only is time consuming due to the delay associated with starting a new pass. Additionally, the parallelism in every stage might be too low to fully utilize the GPU. This fact is also reflected in the performance measurements performed by Sunden *et al.*, which shows that squaring the number of pixels only halves the performance. This means that with a smaller number of pixels per slice, the GPU was not utilized fully.

9.3.1 Increasing Parallelism

In a first step, to increase the parallelism per slice, we decided to alter the assignment of threads to available tasks. Instead of using one thread per ray, we use multiple threads per ray, establishing a cooperative execution scheme. We use all n threads assigned to a ray to concurrently take consecutive samples along the ray, similarly to a windowing approach, as shown in Figure 9.3(b). Then we locally perform hierarchical reduction to compute the attenuation along the viewing ray, before we move the window by n samples. At the same time, we use the samples to update the corresponding n values in the shadow slice. In this way, we increase the available parallelism by the factor of n, but adding a small overhead of a local hierarchical reduction. While this alternative sample scheduling strategy alleviates the problem of underutilization, the overhead of synchronization still remains.

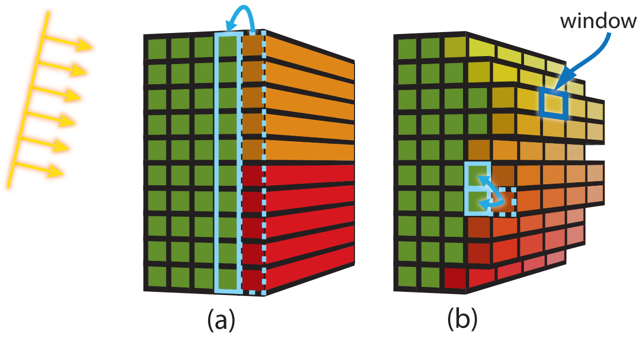
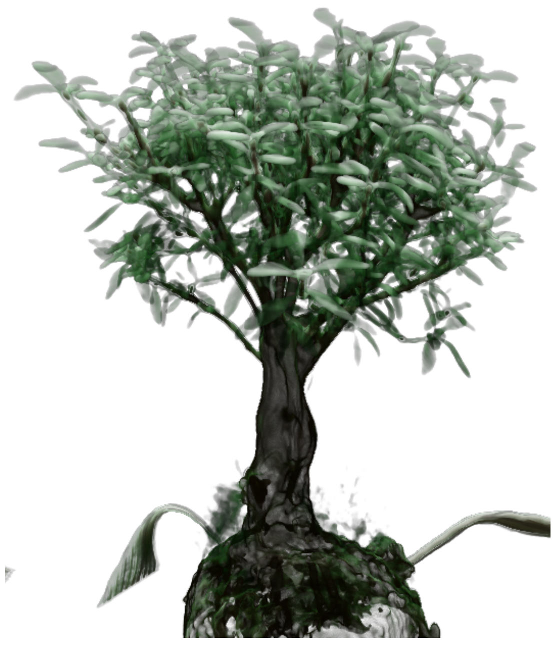


Figure 9.3: The original image plane sweep volume illumination algorithm (a) works on one pixel slice after the other. The dependency between slices (blue arrow) comes with the need to synchronize via the CPU after each slice. Assigning one thread per ray creates the additional problem of GPU underutilization, as only few thread blocks can be started per slice (red and orange). In our approach (b) we assign one thread block of n threads per ray, advancing the sampling window with a step size of n. Additionally, we schedule individual rays rather than slices, reducing the dependencies to individual rays. With these two measures, we can significantly increase utilization and performance.

9.3.2 Avoiding synchronization

To avoid the synchronization via the CPU and to allow concurrent execution of rays from different slices, we build on Softshell's ability to dynamically spawn threads on the GPU. Instead of keeping one slice to read from and one slice to write to, we use a slice cache of depth two. We start the rendering process by launching thread groups for the first slice.


(a) The Bonsai dataset

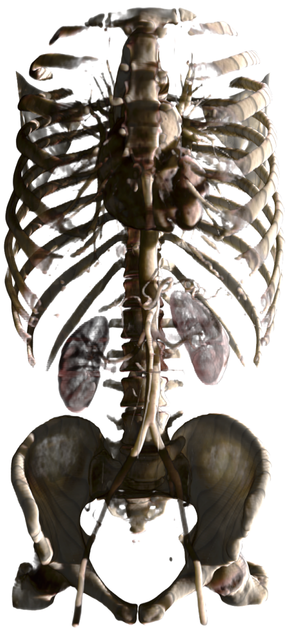

(b) The MECANIX dataset

Figure 9.4: The Bonsai dataset has many small leaves, which can increase thread divergence. The MECANIX dataset is a challenge for efficient GPU execution, as it contains differently sized structures of varying opacity.

Upon completion of each ray, we determine if enough data is available in the cache, so that the immediate successors of that ray can be spawned. Depending on the light orientation, each ray may have one, in case of the light direction being orthogonal to the slice direction, or two predecessors, for arbitrary light directions. If a ray has two predecessors, we use an atomic counter to model the dependency and only start successor rays if their counter has been increased to two. In this way, we allow an arbitrary shaped execution front to move over the image, as shown in Figure 9.3(b). In order not to overwrite information that might still be needed, we alternate writing to the two cache slices. Using this scheduling strategy, rays from different slices can be chosen to fill up free execution units, leading to a higher utilization and a better performance.

Results To achieve a meaningful performance comparison, we use our own reference implementation of Image Plane Sweep Volume Illumination in CUDA and compare it to our scheduled version. Both approaches carry out the same number of operations and only their scheduling differs. In Table 9.2 the render timings for the two tested data sets with different resolutions are shown. Again, we see that the GPU is strongly underutilized for the original approach, as increasing the resolution in direction parallel to the slice does not influence performance. Especially for lower resolutions our scheduled version achieves significant speedups (up to 25 times), which is due to the fact that the underutilization of the GPU is most noticeable in these cases. But even for higher resolutions, our approach is three

Table 9.2: Relative rendering times for different resolutions for the MECANIX and Bonsai dataset rendered with Image Plane Sweep Volume Illumination. Light direction is $-y$. Rows correspond to image resolution in y while columns denote the resolution in x. Note how the base line implementation in CUDA hardly changes with the x resolution due to underutilization. Our scheduled version however, introduces parallelism within rays and softens the dependencies between individual slices, leading to speedups of up to 29 times.

MECANIX

	CUDA			Scheduled		
res	64	256	1024	64	256	1024
64	1.00	1.02	1.01	0.04	0.07	0.17
128	1.97	2.03	2.18	0.07	0.16	0.38
256	4.01	5.21	5.20	0.13	0.31	0.96
512	7.95	10.59	10.97	0.26	0.60	2.44
1024	15.75	20.05	22.23	0.53	1.27	5.09

Bonsai

	CUDA			Scheduled		
res	64	256	1024	64	256	1024
64	1.00	0.99	0.91	0.04	0.06	0.13
128	2.01	2.21	1.95	0.07	0.13	0.29
256	3.74	4.85	4.22	0.13	0.31	0.75
512	7.35	9.94	9.69	0.26	0.65	2.06
1024	14.44	19.68	19.86	0.51	1.28	4.91

times faster than the original. We mainly attribute that to avoiding the synchronization of the individual slices.

9.4 Particle Based Volume Rendering

PBVR [21, 117] is an object-order method used to efficiently render unstructured grid data, such as tetrahedral grids. A dense field of light-emitting, opaque particles is generated inside volumetric datasets, which resembles a probabilistic discrete sampling of the volume. The particle density on the cell-level is chosen proportionally to the opacity of the applied transfer function, while the total amount of particles relates to the cell sizes. Volume rendering is performed by simulating light emission of single particles with respect to mutual occlusion relative to the image plane.

9.4.1 Algorithm overview

To avoid excessive memory consumption, the particle field is generated on the fly. As all cells are independent, an intuitive choice is to parallelize the particle generation and projection on the granularity of cells by launching one thread per tetrahedron. At first, we compute the desired cell opacity by averaging the vertex scalar values and looking up the transfer function. Using the opacity we determine the number of particles to be generated. In the next step, each particle is individually generated by randomly choosing its position within the cell and applying an ‘attraction’ term to accommodate for opacity gradients within the cell. For each particle, we again evaluate the transfer function according to scalar value interpolated from the cell vertices. Finally the particle is projected on the screen. The steps taken for a single cell are outlined in Algorithm 9.

Only a few particles actually contribute to the final color value of a pixel. We use an approach combining the benefits of super sampling and depth buffering. Particles are projected onto sub-pixels while their depth value is used to only maintain the particles closest to the camera along the imaginary ray through a single sub-pixel. Volumetric rendering effects are achieved by compositing the data stored in the sub-pixels into a final color value with respect to their corresponding depth ordering.

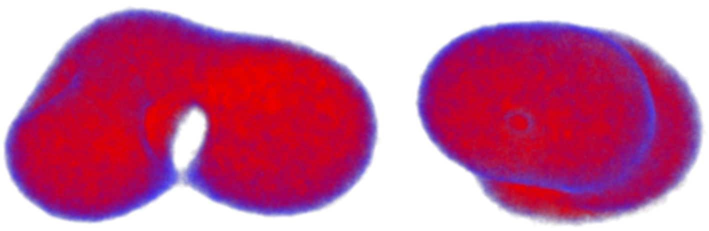


Figure 9.5: Tumor Ablation Simulation with 1.22 million cells in differently aligned view. Due to strongly varying cell sizes (factor 1000), this data set can lead to severe imbalances during rendering.

Algorithm 9: PBVR: particle generation

```
1  $opacity = averge(getOpacities(cellVertices[\ldots]))$ ;
2  $n = drawNumParticles(opacity, cellSize)$ ;
3  **for** $x = 0$ *to* $n$ **do**
4      $b[1 \cdots 3] = random(3)$ ;
5      $b[:] = b \cdot opacities[1 \cdots 3]/max(opacities[:])$ ;
6      $p_x = generateBarycentricPos(cell, b)$ ;
7      $s_x = interpolateScalar(cell, p_x)$ ;
8      $opacity_i = lookupOpacity(s_x, transferFunction)$ ;
9      $projectParticle(p_x, s_x, transferFunction)$ ;
10 **end**
```

9.4.2 Thread divergence during particle generation

The number of particles generated for a particular cell is directly related to the size of the cell and the transfer function. If we consider grids from arbitrary sources, such as physical simulations, inter-volume cell sizes and hence particle counts may vary dramatically. Thus, the loop over all particles (see Algorithm 9) may lead to severe thread divergence.

We again evaluate the divergence arising during rendering. To test uniform cell sizes, with divergence depending on the transfer function only, we use the tetrahedral version of the dragon dataset with 1.25 million cells. The intended scenario, a real unstructured grid, is tested using the simulation data of heat-based radio frequency ablation. The dataset consists of 1.22 million cells, with cell sizes varying between 0.1 and 111.8 volumetric units. Thread divergence during rendering is shown in Table 9.3.

9.4.3 Thread compaction for particle generation

For this use case, we face a situation similar to DVR. Globally, there is a sufficient number of threads executing the same code to ensure high utilization. However, in a local scope, threads are strongly diverging. Thus, we again apply our thread re-convergence strategy and check the number of active threads every L iterations. If the number of active threads drops below $P\%$, we interrupt the execution and try to form non-divergent thread groups via global memory. The only difference in this example is that the recombination needs to only consider threads which still need to project particles, because there are no additional computations to be carried out after exiting the loop.

Table 9.3: Thread divergence observed during PBVR for the Dragon and Ablation dataset, both rendered with 500 million particles. Note that the divergence characteristics are mostly independent of the number of particles and screen resolution and thus the same for all run tests.

	Ablation		Dragon	
method	its	util	its	util
STD	9.50M	17.1%	4.14M	32.1%
RCVG	6.41M	21.6%	3.94M	32.8%
A-RCVG	4.93M	28.8%	3.55M	36.2%

Table 9.4: Render times (ms) for PBVR with and without re-convergence strategies. STD is the baseline CUDA implementation, RCVG corresponds to our re-convergence strategy with $L = 20$ and $P = 0.7$, and A-RCVG is aggressive re-convergence with $L = 1$ and $P = 0.95$. While the Dragon dataset with its uniformly sized cells does not profit from our re-convergence strategies, unstructured grids with differently sized cells like the Ablation dataset can be sped up significantly.

	Dragon					Ablation				
particles	STD	RCVG	speed↑	A-RCVG	speed↑	STD	RCVG	speed↑	A-RCVG	speed↑
$5 \cdot 10^7$	31	35	0.89	45	0.69	95	140	0.68	66	1.44
$1 \cdot 10^8$	56	58	0.97	89	0.64	148	142	1.05	122	1.21
$5 \cdot 10^8$	209	211	0.99	428	0.47	530	148	3.58	577	0.92
$1 \cdot 10^9$	375	371	1.01	848	0.44	927	161	5.75	1186	0.78
$4 \cdot 10^9$	1320	1230	1.07	3562	0.37	3180	243	13.08	5180	0.61

9.4.4 Results

The thread divergence statistics presented in Table 9.3, show that although our re-convergence strategy is able to nearly double the utilization within warps for the ablation dataset, we are still far from 100% utilization. The majority of divergence is caused by a few warps which only have one active thread. The problem with these warps is that they show up at different times during the execution and stay in the divergence list for a long time. If elements remain in the list for too long, we execute them, even if there are no other groups to merge them with, because we cannot know, if any other threads will eventually become available.

In Table 9.4 we present the timings for the two datasets with and without thread re-convergence strategies. For the dragon dataset, with its uniformly sized cells, our re-convergence strategy ($L = 20$, $P = 0.7$) cannot significantly increase utilization or performance. Using an aggressive setup ($L = 1$, $P = 0.9$) we can increase the utilization by 4% and reduce the number of iterations executed by 600k, but the overhead of re-convergence significantly reduces performance. For the ablation dataset, the aggressive setups performs well for low particle counts, for a high number of particles, the overhead again decreases performance. However, non aggressive re-convergence leads to speedups of up to 13 times, especially for high particle counts. That is especially interesting, as the utilization was increased by 26% only. We can only attribute that to additional synergies created by regrouping, *e.g.*, the bus bandwidth can be strongly decreased if threads access neighboring memory locations.

9.5 Discussion

We have shown that GPU-specific issues such as thread divergence or GPU underutilization are inherent to state-of-the art volume rendering techniques. Using advanced scheduling strategies, we were able to increase the performance for particle based volume rendering by a factor of up to 13 times and for image plane sweep volume illumination by up to 26 times (**O1**, **O5**). Although, in general we could increase the warp utilization for direct volume rendering, we could only improve performance for certain data sets, due to the overhead associated with our re-convergence strategies. Using advanced GPU scheduling strategies has the potential to speed up a variety of state-of-the-art volume rendering techniques. We believe that especially complex illumination techniques which involve multiple stages will profit the most from custom GPU scheduling.

Chapter 10

Procedural Modeling

Contents

Open world games such as *Grand Theft Auto*, *Skyrim*, and *Batman: Arkham City* are very successful because they grant players the freedom to explore huge, detailed virtual environments such as cities at their own pace. Procedural modeling has the potential to drastically reduce manual efforts in creating such huge environments, allowing game designers to concentrate their efforts on the elements relevant for narrative and gameplay. However, the grammar derivation of an environment of the size of Manhattan with 100 000 buildings can take up hours, generating billions of polygons and consuming terabytes of storage.

The sheer size of these datasets limits the usefulness of the procedural approach. Rebuilding the environment after parameter tweaking becomes a costly operation, making rapid design iterations impossible. Game levels do not fit in GPU memory and geometry data must be streamed from external storage. Real-time rendering of large models requires visibility culling and levels of detail, making the geometry streaming a complex exercise in real-time data management between CPU and GPU.

To overcome these problems, a combined real-time evaluation and rendering of rule based grammars is needed [107]. We address this need by evaluating the grammar directly on the GPU, generating the geometry that is needed for the current view just in time. A pure GPU approach makes geometry streaming unnecessary, but it requires that all procedural modeling operations are completed within a stringent time budget of a few milliseconds. Unfortunately, there are several reasons why executing rule-based grammars efficiently on the GPU is not straight forward:

R1 State of the art shape grammars require access to local geometric context for features such as occlusion queries and snap-lines. This introduces non-trivial dependencies between rules, and makes parallelization a difficult problem.

R2 The SIMD architecture typical for today's graphics processors only works well on homogeneous workloads. Grammar derivations involving many different rules introduce control flow divergence on SIMD units, easily leading to a performance decrease of more than an order of magnitude.

R3 Individual rules typically only involve a few computations. Even if one can assign homogeneous sets of rules to fully occupy all SIMD units, the overhead of launching computations on the GPU (kernel launches) and writing results back to global memory on the GPU can easily consume 90% of the available time.

R4 Visibility culling or level of detail can usually only be computed after the entire grammar has been evaluated. This is wasteful as geometric primitives are first generated, only to be removed again later using additional computational cycles.

R5 Exploiting temporal coherence is essential to avoid redundant computation. However, level of detail and visibility culling make it necessary to dynamically add or remove geometry. This requires fine grained memory management, which can lead to severe slowdowns on current GPU architectures.

The combination of these difficulties has limited the success of previous attempts at procedural modeling on the GPU. Typical restricted solutions rely on simplified grammars of limited appeal and exhibit poor speed-up over evaluation on the CPU. This chapter presents PGA (Parallel Generation of Architecture), a system that overcomes the above difficulties and is able to procedurally generate and render huge cities entirely on the GPU with competitive performance. Its main contributions are:

- The PGA grammar provides *fully featured modeling capabilities* and is able to create visually appealing, realistic cities. This is not surprising as it is based on a redesign of cga-shape [96] and thus supports mass modeling, evaluation of local geometric context, level of detail and stochastic behavior.
- Despite its expressive power, PGA lends itself to *efficient parallelization* on a SIMD architecture. To this aim, we present a novel GPU based rule scheduler, which exploits parallelism within rules and also groups across rules both locally and globally, yielding significantly faster derivation than previous approaches.
- The design of PGA makes it easy to integrate visibility culling, level of detail and frame-to-frame coherence into the parallel derivation process. As a result, the rendering system supports *real-time walkthroughs* of cities with virtually unlimited size and detail, managing models amounting to billions of polygons.

10.1 Previous Work in Procedural Modeling

Grammar-based procedural modeling. The current state of the art in procedural architecture modeling is still *cga-shape* [96]. The evolution that lead to cga-shape includes Stiny's original *shape grammars* [132], *set grammars* [133], and split operations for façade modeling [147] combined with transformation operations from *L-systems* [110]. Occlusion queries and snap-lines in cga-shape provide the ability to control grammar derivation based on local geometric context. Environmental influence is also exploited using synthetic topiary [109], user designed curves [111], guided derivations [7], interconnected structures [61], vector field guidance [75], and self-sensitive L-systems [104]. Other noteworthy extensions to grammar-based modeling are more general terminal symbols [62] and mesh refinement [39]. There are a number of alternatives to grammar-based modeling that are also able to generate procedural models of high quality [73, 76, 90].

Degree of parallelism in grammars. L-systems allow to compute the transformation for each symbol in a string in parallel, implying an average degree of parallelism proportional to the length of the string $\overline{S}$. Context sensitivity in L-systems is usually defined on the neighbors in the symbol string and does not affect parallelism. For architecture, split grammars are preferred over L-systems [147]. Split grammars use tree-shaped derivations. Because of parent-child relationships, the degree of parallelism is again proportional to the average number of shapes on one level $\overline{S}$. Context-sensitivity in cga-shape operates on a geometric, rather than a symbolic representation. This makes the approach more expressive, but also requires strict rule priorities. Therefore, the standard formulation of cga-shape does not allow for parallel evaluation.

GPU-based grammar evaluation. Their implicit parallelism makes L-systems and split grammars attractive for parallel evaluation. This can be done on CPU clusters [152], but clearly, a GPU is more attractive for this purpose as it provides inexpensive parallelism and the ability to render results directly. However, even the task of mapping a simple L-system to a SIMD architecture produces difficulties. For example, a recent L-system generator [79] for the GPU requires multiple expensive kernel launches per iteration (**R3**) to count the symbols, compute the symbols' output positions and execute the actual rewrite. Moreover, neighboring elements in the string usually map to different rules, so assigning each symbol to one thread leads to thread divergence (**R2**). As a result, GPU evaluation of context-sensitive grammars on the GPU was reported to be even slower than on the CPU. Lacz and Hart [66] evaluate split grammars using vertex and pixel shaders combined with a render-to-texture loop (**R3**). The main load in their system comes from global sorting of intermediate symbols. A similar grammar is evaluated using multi-pass rendering and GPU stream output by Magdics et al. [81]. They reduce divergence by employing a different shader for each output symbol, but still require many rendering passes (**R3**).

Multi-pass rendering can be circumvented by using a fixed-size stack [83]. However, this again leads to divergence (**R2**) and limits parallelism to the number of buildings. More severely, the stack size limits derivation complexity and makes it necessary to rely on instancing of detailed polygonal stock models for terminal shapes. Neither geometric context nor stochasticity is supported. Table 10.1 compares the above approaches to PGA.

Finally, non-parallel grammars can still be evaluated in parallel per pixel, for example, façade textures [35, 84]. However, this approach is highly redundant for neighboring pixels. Similarly, rasterization can be exploited to prune invisible terminals [60].

10.2 Parallel generation of architecture

The key factor for fast rule derivations on the GPU is parallelism. For maximum parallelism, our parallel shape grammar for architecture (PGA) extends the rule set of cga-shape in two distinct ways. First, we introduce a stage-based evaluation model, laying the foundation for the parallel execution of different rules while preserving context sensitivity. Second, PGA enables parallelism on the level of individual rules, tailoring it to an execution on SIMD architectures.

Table 10.1: While cga-shape has a high expressiveness and supports arbitrary context queries between rules, no parallel implementations of cga-shape exists. Previous parallel grammar implementations hardly support context sensitivity, limit the maximum depth of the execution tree/number of rewrites, and do not allow to express complex buildings (**R1**). Although PGA supports cga-shape-style context-sensitivity, our approach shows the highest degree of parallelism (**R1**), optimizes for the SIMD model (**R2**), only requires one kernel launch per stage (**R3**), and reduces the writes to slow global memory (**R3**). For a large city one can observe: $T > \overline{S} > B > D > R$.

method	context sensit.	rewrites	parallel.	launches	SIMD control	mem. writes
cga-shape [96]	geometry	∞	1	-	-	-
Magdics *et al.* [81]	n.a.	∞	$\overline{S}/R$	$D \cdot R$	sort to bins	$\overline{S} \cdot D$
Lipp *et al.* [79]	symbols	∞	$\overline{S}$	$4 \cdot D$	n.a.	$4 \cdot \overline{S} \cdot D$
Marvie *et al.* [83]	n.a.	< 8	B	1	n.a.	T
PGA	geometry	∞	$\approx 4 \cdot \overline{S}$	3	grouping	$\approx \overline{S} \cdot D/32$

$\overline{S}$ average symbols on level; D depth of tree/rewrites; R rules; B buildings; T terminals

10.2.1 Stage-based parallel rule evaluation

In cga-shape, dependencies among rules owed to geometric context are expressed as priorities. For instance, walls must be created before windows, so that windows facing occluding walls can be suppressed. Rule priorities make it very simple to determine serial execution by sorting, but for parallel scheduling, we are interested in efficient identification of *independent* rather than dependent rules (**R1**).

Therefore, PGA replaces the notion of priorities with stages. Rules within one stage can be evaluated in parallel; synchronization is only necessary for the beginning of the next stage. Every rule is assigned one stage and is only allowed to generate symbols for the same or a following stage. This approach differs from cga-shape by explicitly representing independence among the rules contained in each stage. Every stage supports geometric queries concerning neighboring entities to allow for context-sensitive evaluations. We have found a three-stage model consisting of city layout, mass modeling and facade details to be sufficiently expressive.

Sibling queries PGA allows rules to trace information created by predecessor rules, even from upstream stages and including queries to siblings. E.g., a wall tile can query if it is located at a corner by checking how many neighbors it has. The rule creating the tile also creates all its neighbors and can supply this information to the query.

Consistent-evaluation queries Geometric context queries usually target shapes in close proximity, but these shapes are not necessarily limited to predecessors and siblings in the symbolic derivation. However, we can still create an efficient query mechanism by exploiting the fact that close-by shapes will likely correspond to close-by symbols in the derivation. To this aim, we analyze the possible generation paths for the shape being queried and store information about the last common predecessor with the querying symbol during compilation. When the query is being executed, we reconstruct the generation process from the common predecessor, yielding the correct result irrespective of whether the queried shape has already been generated.

10.2.2 Parallelism within rules

For the sake of parallelism, we can go further than just executing multiple rules in parallel. By restructuring operations and masking results by arithmetics, we avoid branches and enable SIMD execution *within* a rule. Multiple threads for one rule reduce thread divergence (**R2**) and significantly increase the number of available threads, yielding better utilization.

To enable compile-time optimizations, PGA requires the thread count for a rule to be known before execution. The thread count can mostly be determined automatically. For example, a split rule creating four elements will work well with four threads. When the thread count depends on the input data, such as the number of floors depending on the building height, runtime profiling can be used to determine a suitable thread count. Finally, the thread count for a given rule can be specified manually.

Notation We extend the cga-shape rule notation with the following parallel construct:

$$\text{predecessor} : \text{cond} \xrightarrow{\text{tc}} \text{successor} : \text{prob},$$

where *predecessor* corresponds to a symbol that will be replaced with *successor*. *Successor* can contain a combination of different operators, which are evaluated by *tc* threads in parallel. The rule is chosen with probability *prob* if condition *cond* holds. If no thread count is specified, PGA will choose an appropriate number.

Parallel operators Most work-intensive operators in shape grammars can be parallelized similar to the following examples (for a more detailed description, see supplemental material).

Repeat fits as many parts into a shape as possible, using one thread per output shape. In the following example, eight threads are used to split a skyscraper into floors of *floor_height*.

$$\text{skyscraper} \xrightarrow{8} \text{RepeatY}\{\mathit{floor\_height} : \text{floor}\}$$

```
template<class OutShape, class InShape>
void RepeatX(InShape shape, float width)
{
  int N = shape.size.x / width;
  for (int i = tid; i < shapes; i += tc)
  {
    float3 size = shape.size * (shape.size.x / N);
    float3 pos = shape.pos + i * size.x;
    generate<OutShape>(shape, pos, size);
  }
}
```

The implementation of Repeat follows a SIMD model. In every iteration, tc new shapes are generated in parallel. Only the last iteration can lead to divergence if N/tc is not integer.

Component split accesses faces or vertices of a volumetric shape with one thread per element.

$$\text{floor} \xrightarrow{4} \text{SplitSidefaces}\{\text{façade}\}$$

To avoid thread divergence, the operator uses rotations and multiplications rather than if-statements to select faces and vertices:

```
template <class OutShape>
void SplitSidefaces(InShape& shape)
{
  float3 w = rotateY(tid * 0.5f * pi) * float3(1,0,0);
  float3 pos = shape.pos + shape.size * w;
  float3 size = shape.size * (float3(1) - w);
  generate<OutShape>(shape, pos, size);
}
```

Non-parallel operators Operators such as geometric transformations (like translation and rotation), shape modifiers, color transformations, random numbers, conditionals and shape definitions cannot be implemented using multiple threads. However, they can still be executed in parallel when sequenced with parallel operators, *e.g.*, rotating the outcome of a component split for generating a windmill:

$$\textit{sails} \xrightarrow{4} \text{SplitSidefaces}\{\text{RotateZ}\{20^\circ : \textit{sail}\}\}$$

To minimize per-rule overhead, it is most efficient to combine parallel and non-parallel operators into a single rule. Note that the same amount of parallelism is available in a combined rule as in separate rules. In the windmill example, a single rule is evaluated by four threads. Splitting the example into one sideface selection and one rotate rule, four threads could be used to execute the sideface rule and one thread to execute each of the four rotate rules, yielding the same amount of parallelism at five times the overhead.

Example We demonstrate the advantage of assigning multiple threads to a rule on a simple building. We split each façade into floors. Each floor is split into tiles, each containing a window and surrounding walls. At street level, we add a door in place of a window (Figure 10.1).

$$\begin{aligned}
\text{façade} &\xrightarrow{2} \text{RepeatY}\{2.5 : \text{floor}\}\\
\text{floor} : \text{at(street)} &\xrightarrow{3} \text{SubdivX}\{2.2 : \text{door}, 2.2 : \text{tile}, 2.2 : \text{tile}\}\\
\text{floor} : \text{!at(street)} &\xrightarrow{2} \text{RepeatX}\{2.2 : \text{tile}\}\\
\text{door} &\xrightarrow{4} \text{Border}\{0.5 : \text{wall}, \epsilon\}\\
\text{tile} &\xrightarrow{4} \text{Border}\{0.5 : \text{wall}, \text{window}\}\\
\text{wall} &\xrightarrow{2} \text{Geometry}\{\text{brick}\}\\
\text{window} &\xrightarrow{2} \text{Geometry}\{\text{glass}\}
\end{aligned}$$

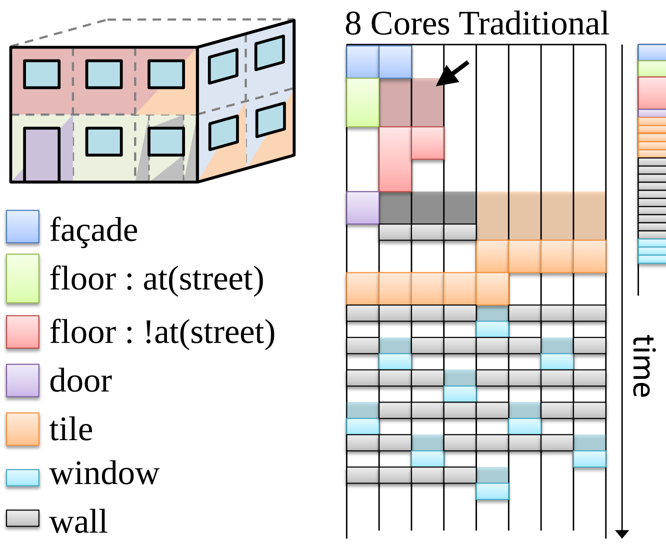


Figure 10.1: (left) A very simple building is specified with seven rules. If a single thread is used to execute each rule (center), most SIMD cores are unoccupied, and divergence further delays execution (darker color, see arrow). (right) With parallel rule evaluation and grouping in PGA, SIMD units can be utilized better, and divergence is avoided, leading to more efficient execution. Note that the same amount of work (colored area) is done in both cases, but PGA achieves better scheduling.

10.3 Rule scheduling

The design of the PGA grammar allows for a high degree of parallelism (**R1**), but additional measures are needed to address thread divergence (**R2**) and minimize the number of kernel launches (**R3**). To ensure consistent nomenclature, we again assume a stream processing model in the following discussion. Note that a shader-based implementation would face the same problems.

10.3.1 Reduction of launches

A naïvestrategy for deriving grammars on the GPU is to keep a set of S active symbols in GPU main memory. A kernel with S threads is started to execute the corresponding rules, writing S_{new} newly produced symbols to memory. When the kernel finishes, S_{new} is read back from the GPU, and a new kernel is launched. This approach requires synchronization between kernel launches for every step of the derivation, which stalls the GPU for 50μs and leads to low performance.

With Softshell we overcome this issue. Each of the three PGA stages corresponds to one megakernel launch. In each state, all threads retrieve symbols from the work queue and execute rules continuously, until all symbols for this stage have been processed. Using the thread aggregation methods homogeneous workloads can be assigned to all threads of a group. Newly generated symbols are inserted into the queue if they should be executed within the same stage. If symbols for a following stage are created, we insert them into separate queues and in this way delay their execution for the next megakernel launch. In this setup the number of kernel launches is always equal to the number of stages.

10.3.2 Rule grouping

A key factor for fast rule derivation is avoiding divergence (**R2**), which requires assigning symbols of the same kind to all threads of a warp. Sorting symbols in global memory can become a major bottleneck due to memory latency, taking up to $0.5ms$ for about 500 symbols [66].

Fortunately, we do not need to sort all symbols to generate groups of a sufficient size. Due to parallelism within rules, combining just a few symbols is usually sufficient. In Softshell we can use a hash map in global memory to automatically store a set of active symbols for each rule. A new symbol is only inserted directly into the queue if it can be executed divergence-free, *i.e.* the symbol uses a number of threads that are a multiple of the warp size. Otherwise, it is inserted into the hash map. The hash map is scanned periodically. If any rule has a sufficient number of symbols ready to be scheduled, we transfer them to the queue (Figure 10.2). Softshell also schedules symbols for execution if the work queue runs low on entries or if symbols have been stored in the hash map for too long.

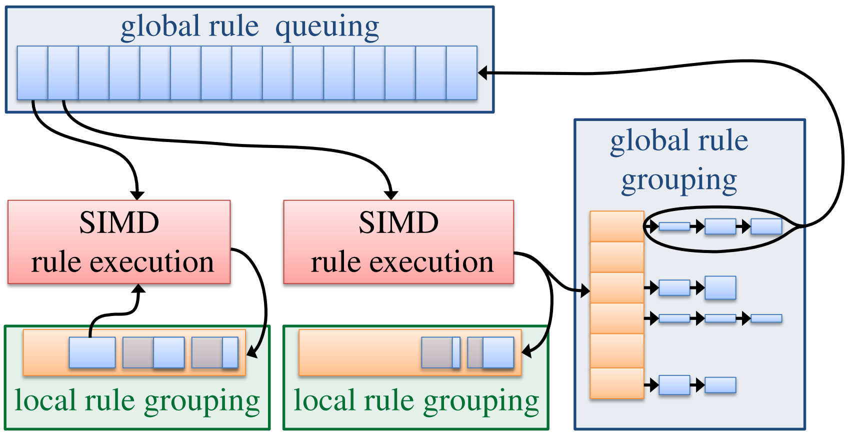


Figure 10.2: To efficiently manage symbols awaiting evaluation, we use a two level approach. Global queuing and grouping (blue) employs a hash map to merge rules of the same type before they are inserted into the execution queue. To avoid the costly transfer to global memory, we introduce a local grouping mechanism (green). Each block keeps a list of a few rules in local shared memory. These rules are executed before new ones are drawn from the global queue. If local grouping runs out of memory or if there are not enough rules available, we resort to global grouping.

10.3.3 Local grouping

Although the rule grouping is orders of magnitude faster than global sorting, the overhead of writing symbols to global memory can still be significant (**R3**). Therefore, we first group symbols in local shared memory. As shared memory is limited and there might be hundreds of different rules to be scheduled, we cannot keep a queue for each rule. Instead, we use our dynamic shared queue and allocate new sets as needed. Each set has a header with rule type, number of entries and time stamp. When a new symbol is created, we scan through shared memory and search for a set of the given type to add the symbol to it.

In every iteration, we check shared memory first. Any complete set matching the current stage is immediately removed from shared memory and executed. Otherwise, elements are drawn from the global queue. If many complete sets are found or if shared memory is full, we move them to the global queue. Incomplete sets are also moved to the global queue after a certain time, so they can be merged with symbols from other SIMD units.

This rule scheduling has several advantages: First, the number of reads and writes to global memory is minimized. Second, if local structures are full and we need to spill to global memory, the resulting writes are highly coalesced, leading to very efficient memory bus usage. Third, shared memory is nearly equivalent to registers in terms of speed, while still allowing to feed coherent workloads to thread blocks. And fourth, multiple threads per rule allow sets to be small while still providing work for a whole thread block.

10.4 Real-time rendering of procedural cities

While the proposed rule-scheduling system helps to overcome **R2** and **R3**, generating an entire city in full detail still exceeds GPU memory capacity. However, rendering the current view only requires the geometry of the buildings within the viewing frustum, ideally represented at a level of detail consistent with their apparent size to avoid aliasing.

Thus, we describe a strategy for generating the necessary geometry just in time. We prune the execution tree where needed (**R4**), taking advantage of view-frustum and visibility culling and dynamically insert terminal symbols to generate level of detail (LOD). When the view is changed, new buildings come into sight and LOD should be adjusted. In such a case, parts of the geometry need to be re-generated. However, most of the geometry can be re-used by exploiting frame-to-frame coherence (**R5**).

We discuss the combined generation and rendering process along the three stages of our model (Figure 10.4).

1. **City layout and building hulls** For the city layout, we use GIS data to specify building lots or generate street layouts and parcels procedurally. For each parcel, we also create a boundary volume (hull).

2. **Building specification** Given the hull of a building, additional constraints and parameters concerning the building are specified. This information must be sufficient to enable context-sensitivity in the following stages.

3. **Building construction** In the final building construction stage, rules can be executed as they become available, allowing a maximum degree of parallelism. Therefore, the rules should only rely on the information that has been defined in previous stages.

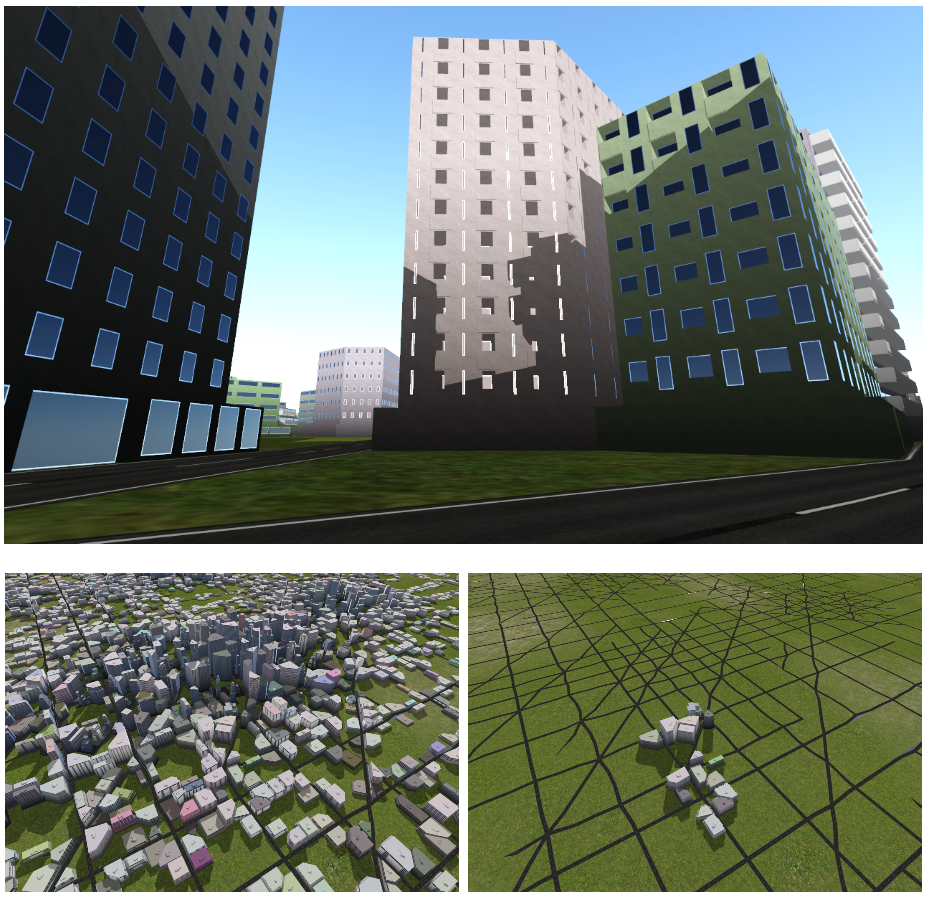


Figure 10.3: (top) Street-side view of a city. Using a combination of frustum culling, visibility culling, and level of detail, the amount of generated and rendered geometry can be reduced without affecting the final image. (bottom left) The full scene consists of 5869 buildings, 10M rules, and 23M triangles. (bottom right) The reduced geometry has 17 buildings, 46k rules and 55k triangles.

10.4.1 City layout, hulls and frustum culling

The purpose of this stage is to define where buildings should be generated and which volume they are allowed to occupy. Initially, the user specifies a single rule. The input data for this rule can be provided by loading GIS data or by procedurally generating cells. The initial rule is called for every building footprint in the GIS data set. If no footprints have been loaded, the world is split into a grid, and the initial rule is executed for each grid cell within a user specified radius around the camera.

Building hulls For the output of the first stage, we use so called building hulls: an intermediate structure describing the space a single building is allowed to fill. Building hulls can be used as efficient query objects in the building specification stage, so-called building representatives.

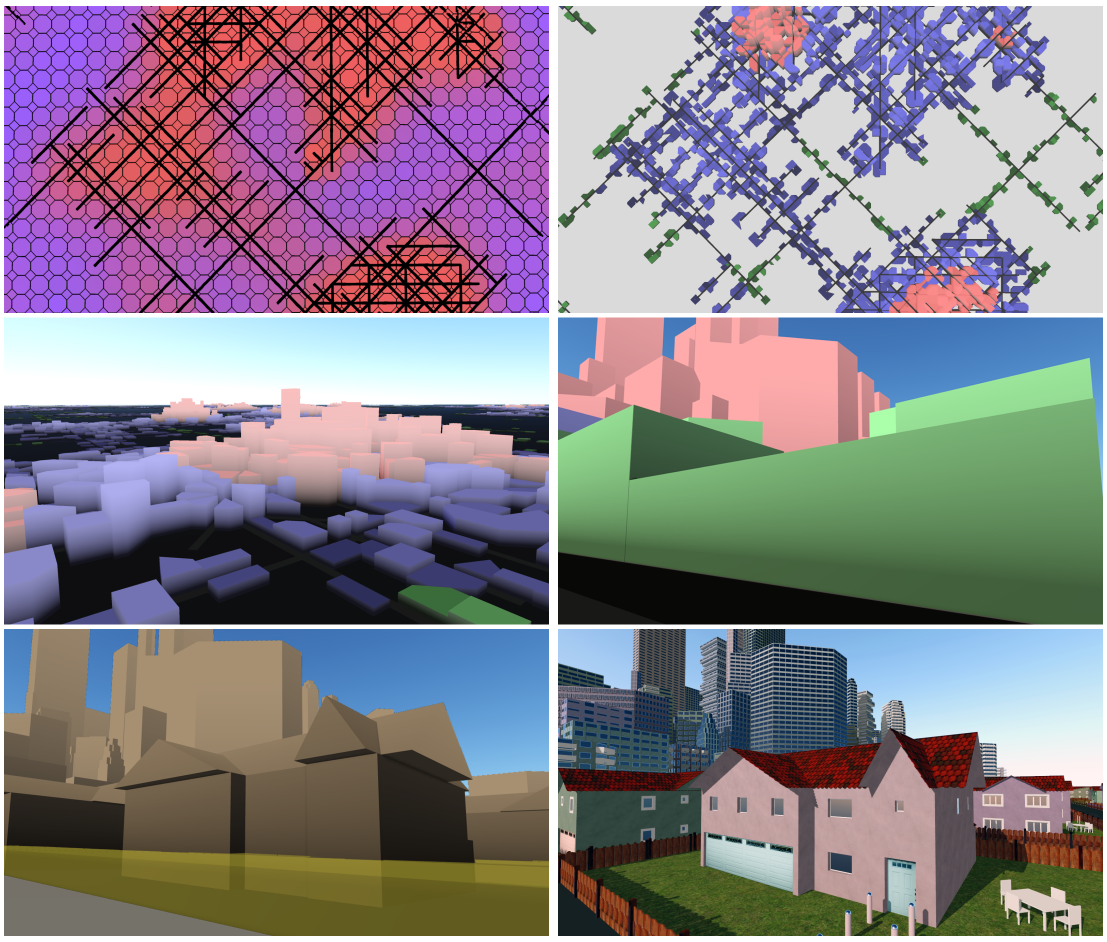


Figure 10.4: Splitting the rule derivation into three stages facilitates context-sensitive evaluation while aiding parallelism. In the first stage (top), the street layout is set up and (center) the maximal bounds of buildings are specified as hull models. The hull colors correspond to different building classes. (bottom left) The output of the second stage are detailed mass models, where brown encodes *opaque* structures and yellow corresponds to *enclosing* structures, *e.g.*, structures covering the fence around the house. (bottom right) During the final stage, the entire building geometry is constructed.

View-frustum culling If a shape has been marked as a building representative, it is automatically subject to view frustum culling when its rule is being executed. To implement view frustum culling, we apply the trivial reject test of the Cohen-Sutherland line clipping algorithm in 3D to all vertices of the shape: If all vertices are outside of the same side of the frustum, the shape is ignored. We parallelize this step using eight threads per shape and combine the voting masks for all threads using a hierarchical reduction in shared memory. If a building hull is not visible, the entire building need not be derived, skipping all following stages.

10.4.2 Building specification and visibility culling

The purpose of the building specification stage is to define the outer geometry of a building as accurately as possible, so that sufficient information for the following rules is available. This process is generally known as mass modeling. To specify the mass model, we again rely on the building representative, which can be accessed in all rules for the building.

When a shape is added to the mass model, additional information can be attached to it. One additional information item is the shape's visibility state, which can either be *opaque*, *enclosing* or *hidden*. Opaque shapes are expected to be fully filled by the building and do not allow a viewer to see through. Enclosing shapes will contain visible shapes, but can be looked through from some viewpoints. Hidden masses will not contain any visible shapes. These properties are important for queries (Figure 10.5): To get the right results from queries, the entire mass model must be specified. We expect the building specification to be completed when a building is moved to the next stage.

Snap-lines Like cga-shape, we also allow the creation of snap-lines. Operators like Repeat or Subdivide automatically snap their splits to the snap-lines specified for the building. Using snap-lines, elements among different mass model parts can be aligned (Figure 10.5). To efficiently implement the concept of snap lines, we use the building specification stage. During this stage, snap-lines can be added to buildings, but not queried. When the building is moved to the next stage, we build an accelerated search structure for fast queries.

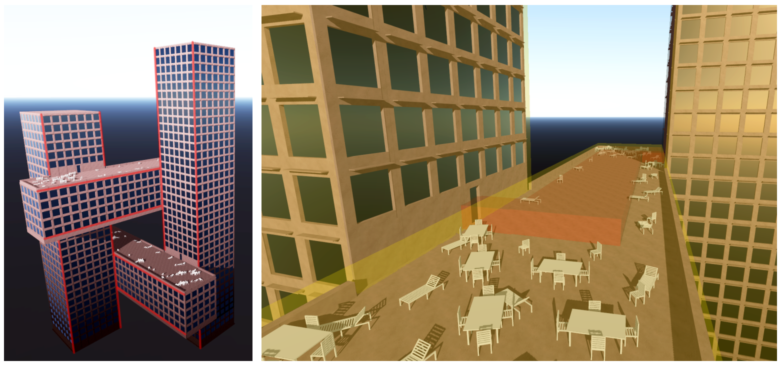


Figure 10.5: Model of the Cross-Towers being built in Seoul. (left) Using snap-lines, the floors and window of the individual tower parts are aligned, creating a coherent façade. (right) Different visibility states used in mass modeling: In addition to the opaque tower shapes (not depicted), enclosing boxes (yellow) are added around the towers to include the fly sheets and on top of the bridge to capture the assembly of outdoor furniture. A hidden corridor is set up between the doors (red). During rule derivation the outdoor furniture is tested against this corridor, avoiding an obstruction of the walkway. Accordingly, the façade tiles are transformed into doors if they intersect with the corridor.

Visibility culling Especially in city walkthroughs, large portions of the urban environment can be removed by visibility culling. Using the information provided during mass modeling, we can implement visibility culling after the building specification stage is finished. Since the use of hardware occlusion queries would require to render every building individually and to switch back and forth between rule derivation and queries issued from the CPU, we use a custom hierarchical depth buffer [32]. When a building is ready to move to the next stage, the mass model shapes are queried against this depth buffer. If a shape flagged as *opaque* or *enclosing* is determined to be visible, the building is constructed, and we update the depth buffer with the maximal depth of the *opaque* shapes. If no part of the building's mass model is visible, the entire building can be skipped.

To efficiently query and update the depth buffer, we divide the initial level of the buffer into 8×8 cells and assign one thread to each cell. For testing a shape against the buffer, we traverse the sub-tree associated with each cell and compare the cell's depth to the minimum depth of the shape. For checking if a shape surrounds a cell or a cell is contained within a shape, we exploit the fact that all our basic shapes are convex. We project each shape's vertices to screen space and compute their outline using the parallel Jarvis's march algorithm [51]. Testing the convex polygon against the rectangle cells of the depth buffer is trivial. If the shape contains round elements (cylinders or spheres), we use the circumscribed circle for the depth query and the inscribed circle for the update, implementing a conservative test and update strategy.

For making visibility culling most effective, we process buildings close to the camera first. Our grammar derivation is highly parallel, so we cannot enforce any ordering on the buildings' processing order. However, it is still possible to initiate the execution in a way that favors closer buildings. In the city layout and building hulls stage, we run through the cells or building outlines (in case of GIS data) in a spiral sequence, starting with the location nearest to the camera. Because the queues are loosely operating in FIFO manner, buildings are approximately constructed in the desired order.

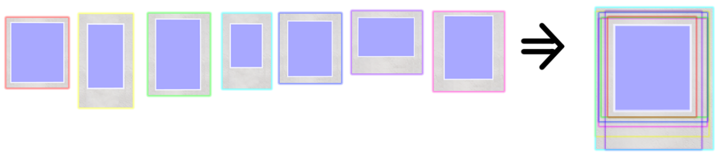
10.4.3 Building construction and level of detail

The building construction stage starts with the rules for the mass model shapes. From this point onwards, all available rules can be executed immediately, because no later synchronization points exist. Occlusion queries and snap-lines automatically provide the data required for the building under construction.

Procedural level of detail PGA allows to stop the evaluation of rules early and insert a terminal symbol, if a point is reached for which further derivations would not visibly enhance the rendering. Because the input shape for each rule approximates the results of the following operations, we use this shape as a terminal for LOD. To produce a result visually similar to the detailed model, we apply textures. These LOD textures are automatically generated in a preprocessing step. There are three issues with this approach: First, the input shape does not always match the shapes produced in later derivation; *e.g.*, if the two dimensional tiles of a façade are augmented with balconies, the shapes do not even share the same dimensionality. Second, the shape's appearance may depend on the input shape size and on random numbers drawn during rule evaluation, and thus a single, precomputed texture can never cover all possible instances. For example, the number of windows on a floor depends on the length of the wall. And third, there might be probabilistic selection among alternative rules.

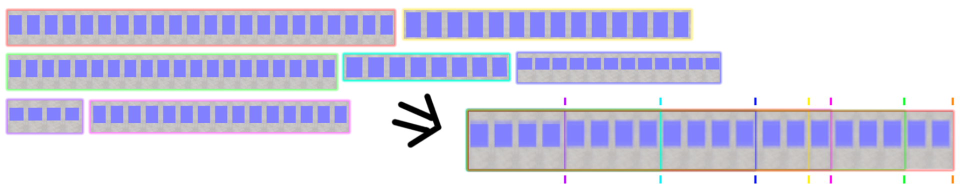
To overcome the problem of non-matching shape boundaries, we rely on user input during rule design. The user can flag each rule as a possible LOD terminal. In this way, LOD shapes that do not match the boundaries of the following rules can be avoided. Furthermore, it is possible to control the number of generated LOD terminals. To deal with the influence of input size and random numbers on the shape's appearance, we analyze the operators used within the rule, create one unified texture and adjust texture coordinates to match the desired appearance as closely as possible. Consider Figure 10.6: First, we automatically create a sufficient number of samples for the rule, varying the input parameters and random seeds. Next, we align the samples. For the tile rule, which creates randomly sized walls with a window in the middle, all windows are aligned. For the floor rule, which creates as many tiles as fit into the scope of the floor, the tiles are aligned. The aligned combination of images is then used as LOD texture. When an LOD terminal is created, the texture coordinates are adjusted accordingly, selecting a fitting border around a window or selecting the right number of tiles.

$$\text{tile} \rightarrow \text{Border}\{rand4(0.1, 0.8) : \text{wall},\ \text{window}\}$$



(a) LOD texture generation for tile rule

$$\text{floor} \rightarrow \text{RepeatX}\{2.2 : \text{tile}\}$$



(b) LOD texture generation for floor rule

Figure 10.6: LOD textures are generated by sampling the input space of the rules. Analyzing the operators used in the rules, the textures for different input parameters can be aligned to generate a single LOD texture per rule. During rendering, the right texture segments are selected. In this way, one LOD texture can be used for different buildings as shown in Figure 10.7.

However, not all rules can be covered by this strategy, *e.g.*, if the position of a door is randomly chosen. In cases where we cannot provide a special strategy for the operator, we simply merge all randomly sampled instances, creating an average among all possible shapes and use it as texture. When creating the LOD textures, we process the rule tree bottom up. Using the terminal rules, we sample the lowest level of LOD terminals. These

LOD terminals are then used in the creation of the next level of LOD shapes. While the textures are created, we also evaluate at which distance LOD should be switched. For this decision, we compute the average pixel difference between the LOD version and the previously generated samples for varying viewing distance and store the information for rendering (Figure 10.7).

10.4.4 Frame-to-frame coherence

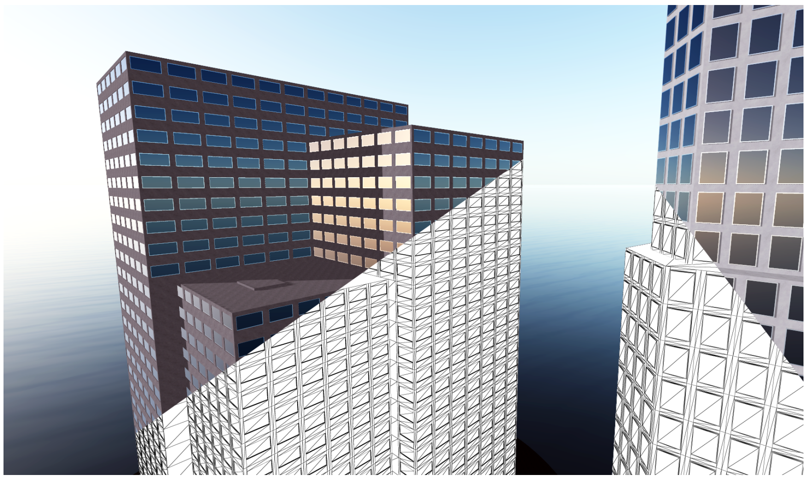
Real-time performance is only possible by using frame-to-frame coherence. If the camera parameters do not change much, most of the buildings with their respective LOD selections can be re-used. The only difficulty is keeping track of the vertex and index buffer.

Sparse LOD rule tree To keep track of entities in the scene, we keep a list of buildings and an associated sparse tree of rules that are potential LOD terminals. In each frame, we run through all buildings, testing their visibility and updating the custom depth buffer. If a building is not visible, because it is too far away from the camera, we remove it completely and rebuild it only when it comes into range again. If a building is occluded by others or out of the frustum, we remove all geometry associated with it, but keep its specification to enable a quick rebuild. If a building is visible, we evaluate its LOD nodes. Should a sufficient number of LOD nodes be ready for change, we recreate the building according to the current view.

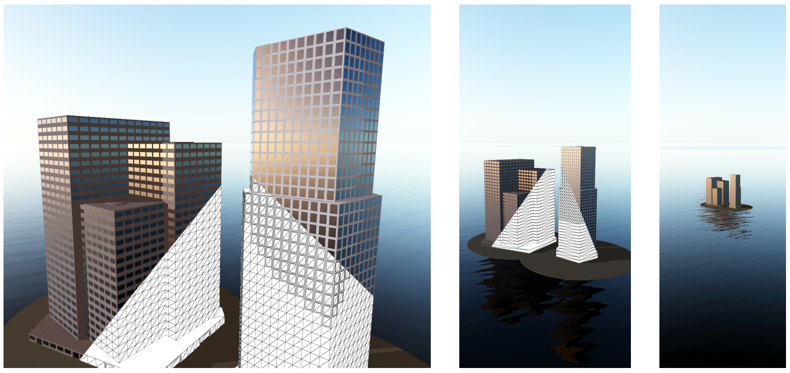
Buffer management To keep the overhead of vertex and index buffer management low and to avoid memory waste, we build a memory management structure directly in the index buffer. We start by allocating a vertex and index buffer with sufficient size, initialized with zeros. Unused parts of the index buffer will thus be ignored by rendering. We partition the index buffer into blocks similar to the buddy memory allocator [58]. With each block in the index buffer, we associate a block in the vertex buffer based on an estimated vertex to index ratio. We choose the smallest block size to be able to hold the lowest LOD version of every building type.

We use the first index in each block as an atomically operated counter to keep track of the number of available indices in each block and the second index as a pointer, creating a list. The third index is a copy of the second, forming a degenerated triangle that is skipped during rendering. For each block size, we keep a list of free blocks. If a new block is requested, we check if a free block of suitable size is available. If this is the case, we remove it from the list. If not, we take a block of bigger size and split it. If blocks are freed, they are inserted again into the respective lists. To counteract fragmentation, we periodically scan the lists, merge neighboring blocks and move blocks with lower indices to the front.

To manage the allocation of vertices and indices during rule derivation, each building keeps an additional list of its index blocks. If a rule requests memory, we try to allocate indices from the front of the list by increasing the counter in the block. If there is insufficient space in the block, or if there is no block in the list yet, we request a block of the next bigger size from the memory manager. If the geometry of a building is removed, we free all



(a) Full detail $176.2k$ vertices, $88.0k$ triangles



(b) $14.8k$ vertices, $7.4k$ triangles **(c)** $2.7k/1.3k$ **(d)** $418/268$

Figure 10.7: Different views on two towers with LOD rendering enabled. Both buildings use the same LOD textures for tiles, floors and façades, although the window size and number of windows per façade differ. Different LOD settings blend seamlessly on the right building in (b) and (c). Also note that context-sensitive rules do not create windows where the parts of the left building meet.

used blocks, inserting them back into the free lists. A separate list of blocks is kept for rules without an associated building.

10.5 Results

To illustrate the influence of the various components of PGA, we compare the performance of multiple versions of our implementation. The base line is a single-threaded CPU implementation of PGA written in C++ (*CPU*), which uses simple sets for the PGA stage queues and is compiled with SSE support. To implement an optimized multi-threaded version of PGA on the CPU (*C-PGA*), we use a custom memory allocator and lock-free rule queues. For the GPU baseline, we provide a CUDA-based implementation of PGA (*GPU*),

Table 10.2: Generation times in ms for different methods and test scenes. *CPU* is our base line C++ implementation; *C-PGA 4* is a CPU version of PGA using a custom memory allocator and lock free-rule queues, executed with four threads; *transf* captures the transfer time from CPU to GPU memory; *GPU* is a baseline CUDA implementation of PGA; *GPU G* uses persistent threads and rule grouping; *G-PGA* additionally uses multiple threads per rule and local rule grouping. *C-PGA* ↑ corresponds to the speedup of *C-PGA 4* over *CPU*. G-PGA↑ corresponds to the speedup of *G-PGA* over *CPU* (with CPU-GPU transfer times added). Because the suburban scene does not fit into GPU memory, we omit the transfer time. For measuring GPU generation time of this scene, we replaced previously generated geometry when the 2GB geometry buffer ran out of memory.

	CPU	C-PGA 4	transf	GPU	GPU G	G-PGA	C-PGA↑	G-PGA↑
Tree4,3	11.7	4.49	0.18	1.57	1.55	0.84	2.6	14.1
Tree8,4	53 546.1	135.40	154.02	56.72	48.87	19.29	395.5	2783.8
Residential	77.8	8.36	1.77	7.38	4.90	2.31	9.3	34.4
Cross-Towers	2092.8	24.12	7.44	19.17	11.86	6.97	86.7	301.3
Skyscrapers	6262.6	545.15	187.68	394.65	263.37	164.09	11.5	39.3
CityOverview	16 573.0	996.33	331.98	708.02	346.73	249.89	16.6	67.6
Suburban	53 605.3	13 115.90	-	11 041.17	6689.11	4551.21	4.1	11.8

using a global, unordered set of ready-to-be-processed rules. For each iteration, a kernel is launched and one thread is used to execute a single rule. Intermediate symbols are allocated using the same dynamic memory allocator as used in the other GPU implementations. Our second GPU implementation is based on Softshell, using persistent threads and rule grouping in global memory (*GPU G*). Finally, we evaluated the full PGA implementation adding rule grouping in local shared memory and multiple threads per rule (*G-PGA*). All measurements were run on the same machine with an Intel Core i7-940 Quad Core CPU (2.93 GHz) and an NVIDIA Quadro 6000 GPU.

10.5.1 Generating a single view

We selected a variety of test scenes, including small and large setups, geometry-heavy and rule-heavy derivations, context-free and context-sensitive rules, scenes with and without level of detail and snap-lines. The characteristics of the scenes are outlined in Table 10.3 and shown in Figure 10.8.

Generation times are summarized in Table 10.2. Our optimized *C-PGA* implementation (four threads) achieves speed-ups of 2.6-395.5 over *CPU*. Increasing the thread count above four reduced the performance, which indicates that the four cores are well utilized. With the full-featured GPU implementation, we achieved speedups between 11.8 and 2783.8 in comparison to *CPU* (with CPU-GPU transfer added). For the bigger test scenes, the geometry generation is faster than the memory transfer from the CPU.

The tree data sets are the simplest test cases, because they only use a few context free rules. We set up a rule system similar to the tree found in GPU Shape Grammars [83], allowing for a direct comparison to their implementation. They report a generation time of $40ms$ for their tree with 240 terminals on a GPU with processing power equivalent to ours. Our tree with 283 terminals is generated in $4.5ms$ on the CPU and $0.84ms$ on the GPU, which corresponds to a speedup of approximately 47. Within a time frame of $20ms$, we can generate a tree with 23 000 terminals and $10M$ triangles. The tree demonstrates that a small number of context-free rules executed a few thousand times do not pose a challenge to our system. In the large tree example, we can generate geometry ten times faster than transferring it from CPU to GPU memory. Local rule grouping of *GPU GM* shows its strength in this example, avoiding global memory access.

Context-sensitive buildings like a residential house or the Cross-Towers are generated in two and seven milliseconds on the GPU, respectively. As a residential house consists of a larger number of different rules than the Cross-Towers, using multiple threads per rule leads to a higher utilization and thus has a greater effect on performance. In combination with LOD (Skyscrapers and CityOverview), *G-PGA* shows speedups of 39 to 67 times, indicating that our implementation can deal well with big cities and different levels of detail. The Suburban scenario is an extreme case, with huge amounts of geometry and a large number of intermediate symbols being created. These intermediate symbols put a lot of pressure on the dynamic memory allocator on the GPU, forcing us to increase the size of the memory pool from 100MB to 300MB. Nevertheless, a reasonable speedup of 11.8 could still be achieved.

Table 10.3: Our test scenes feature up to $37k$ buildings (obj), $800k$ non-terminal symbols (nodes), 6M terminals (term), $300M$ vertices (vert), and $170M$ polygons (poly). We cover different aspects, like context-sensitivity, the use of snap-lines and level of detail. For the tree dataset, the numbers correspond to the iterations and number of tree branches. For the Skyscrapers and CityOverview testcase, we include LOD and non-LOD rendering times. The full resolution derivations contain 600 million and 1 billion polygons respectively. The tree, Skyscrapers, CityOverview and suburban test cases are shown in Figure 10.8. The cross towers are shown in Figure 10.5 and the residential house corresponds to the house in Figure 10.4.

	obj	nodes	term	vert	poly
Tree4,3	-	121	283	3896	5232
Tree8,4	-	9k	23k	6M	10M
Residential house*	1	297	2423	99k	62k
Cross-Towers*‡	1	3327	17k	363k	182k
Skyscrapers*	11k	73M	59M	1.51G	724M
Skyscrapers*†	11k	292k	825k	7.63M	3.82M
CityOverview*	37k	67M	91M	4.45G	2.23G
CityOverview*†	37k	228k	842k	9.93M	5.66M
Suburban*	4244	796k	6.0M	278.8M	167.8M

* context sensitive; † level of detail; ‡ snap lines

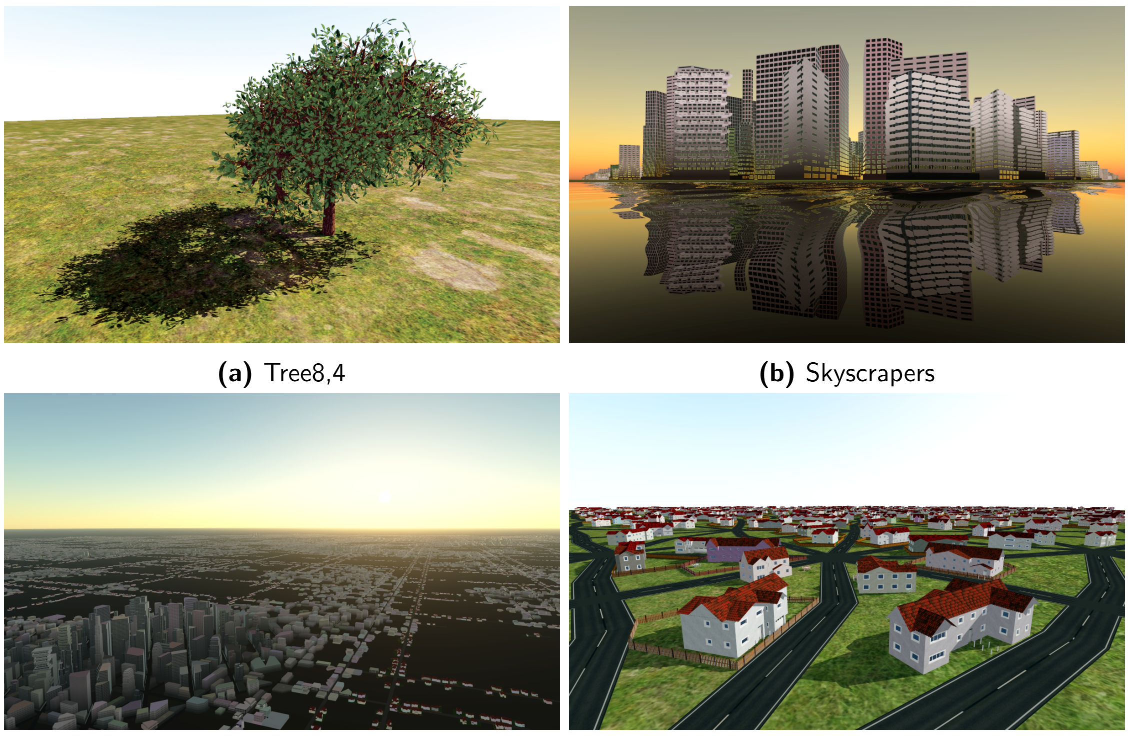

(a) Tree8,4 **(b)** Skyscrapers **(c)** CityOverview **(d)** Suburban

Figure 10.8: Four test scenes: The tree is the only model that exclusively uses context-free rules. The skyscraper scene contains $11k$ buildings with LOD and the camera in the center. The city overview scene contains $37k$ buildings distributed over an area of $80km^2$. The suburban scene contains $4k$ full-detail buildings.

10.5.2 Rendering a continuous movement

To evaluate the effect of culling and frame-to-frame coherence, we used an *infinite city* test case with adjustable clipping distance. Street layouts and terrain were derived procedurally and the city blocks were populated with a combination of suburban, residential and office buildings. As clipping distance we chose 1000 and 3500 meters. For $1000m$ about 3500 buildings are within the visible range; a full detail generation would need $150M$ polygons and $18M$ rules. At a clipping distance of $3500m$, $47k$ buildings are in the visible range; their full detail representation has 2 billion polygons generated by $240M$ rules. The LOD versions for these two scenarios require $1.6M$ and $7M$ polygons, respectively. Our test scenario contains a walkthrough (movement speed $1.5m/s$) of the suburban area, a fast drive to a dense skyscraper area ($20m/s$), rising over the city ($50m/s$) and flying above the city at rocket speed ($300m/s - 100km/s$). Selected frames from the scenario are shown in Figure 10.9. We tested four methods: *C-PGA* deriving the required geometry on the CPU and transferring it to GPU memory, *G-PGA* in a basic version, *G-PGA* with view frustum and visibility culling, and *G-PGA* with culling and frame-to-frame coherence. All methods used LOD.

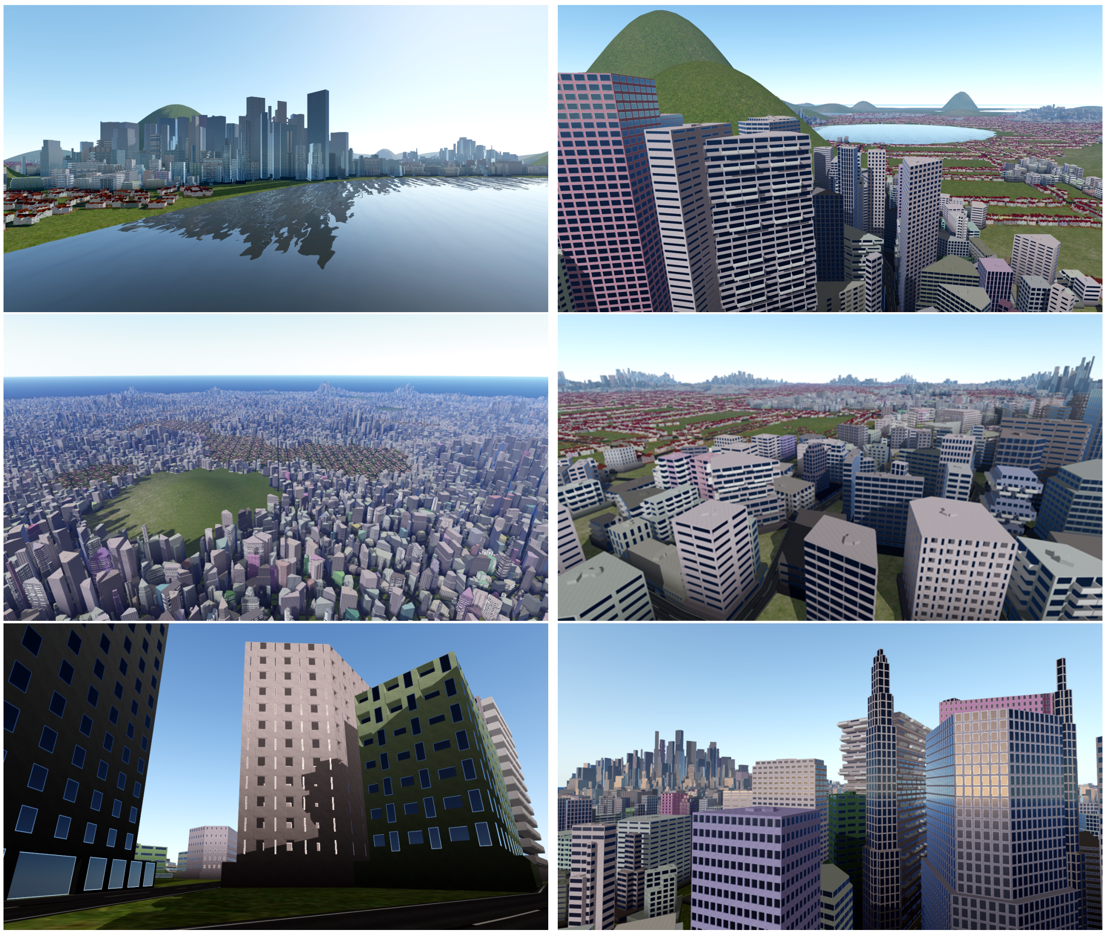
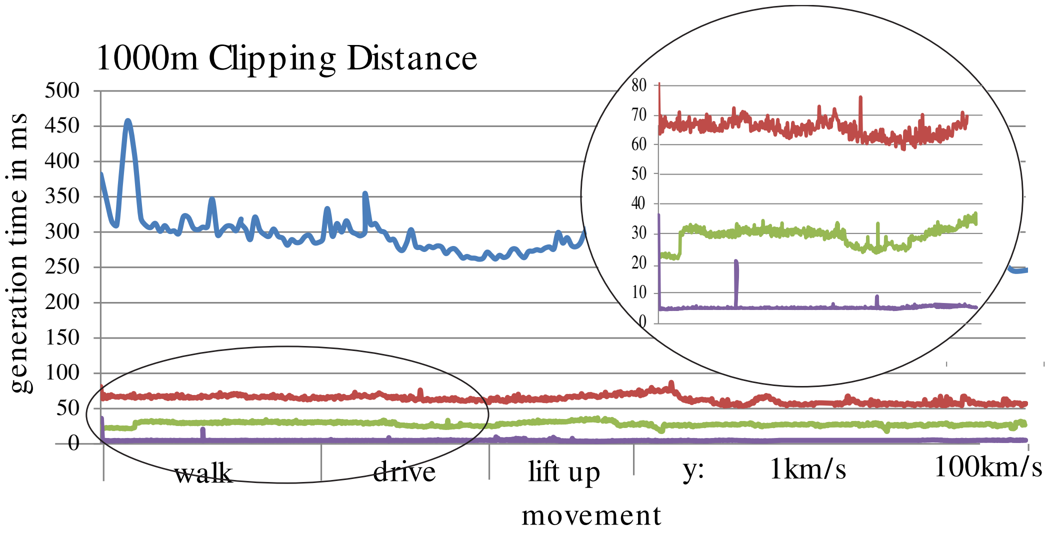


Figure 10.9: Unlimited cities with highly detailed, context-sensitive buildings can be generated in real-time on the GPU using our parallel shape grammar. The visible $28km^2$ of the city contain $47\,000$ buildings. In full detail, these buildings are generated by 240 million rules, evaluating to 2 billion polygons. Generating one view with levels of detail ($7M$ polygons) from scratch takes $500ms$. Exploiting frame-to-frame coherence, we update the geometry in $50ms$ on a standard PC, even if the viewer moves at supersonic speed.

The frame generation times for the methods and two clipping distances are shown in Figure 10.10. For $1000m$ clipping distance, our CPU implementation needs between $250ms$ and $400ms$ per frame. Our basic GPU implementation is about five times faster with an average of $65ms$ per frame. With culling, performance approximately doubles at $30ms$ per frame. If, additionaly, we exploit frame-to-frame coherence, only about $5ms$ per frame are needed after the initial frame, yielding a speedup of about 60 compared to *C-PGA*. For $3500m$ clipping distance, we achieved the following results: *C-PGA* $3500ms$, *G-PGA* $525ms$, *G-PGA* with culling $280ms$, and *G-PGA* with frame-to-frame coherence $50ms$ per frame. Again, our full GPU implementation was still able to create the city in real-time and was about 60 times faster than *C-PGA*.



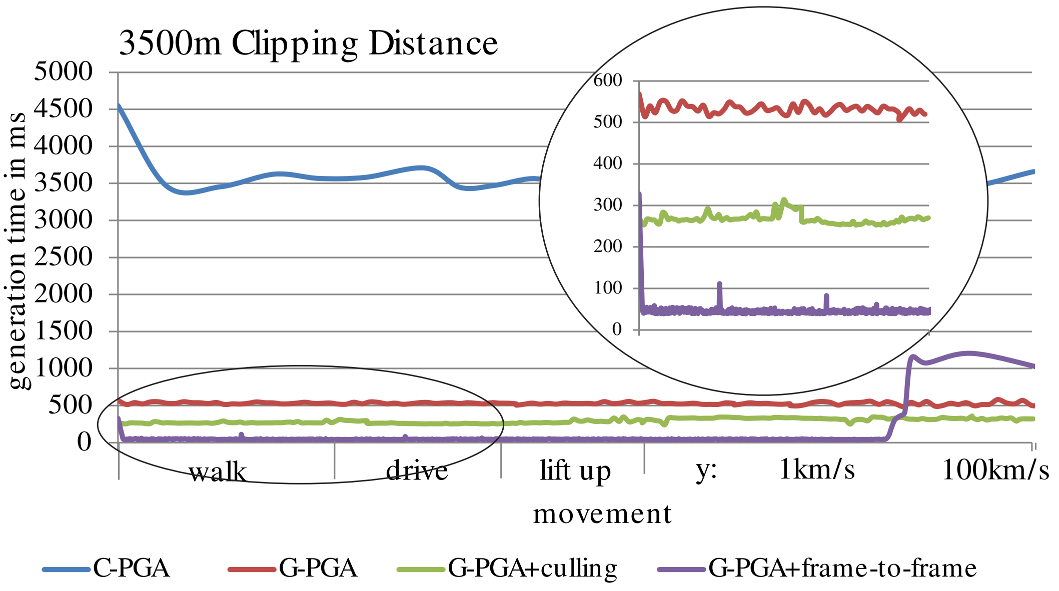


Figure 10.10: Comparison of frame generation times in an unlimited city (Figure 10.9). CPU streaming is too slow for interactive rendering, while our GPU method with frame-to-frame coherence achieves $5ms$ ($1000m$) and $50ms$ ($3500m$) respectively. Frame-to-frame coherence breaks if the camera is rotated fast (spikes in enlarged areas) or if the movement speed is very fast (about 10km/s).

We hardly noticed any influence of the movement speed on the performance of the frame-to-frame coherence method until we increased the movement speed to 10km/s in the $3500m$ clipping scenario, where subsequent frames do no longer have significant coherence, and incremental evaluation is even detrimental on performance. For $1000m$ clipping the frame-to-frame coherence method worked well even for very high speeds. However, if other operations use the GPU simultaneously, frame rate may decrease significantly. A much easier way to break frame-to-frame coherence is camera rotation, which was not considered in the frame-to-frame coherence scheme. We performed two fast camera rotations during the walking phase, leading to spikes in the frame rate. However, the system regenerates quickly.

10.6 Discussion

We have presented a system that is able to parallelize the derivation of shape grammars for GPU execution without sacrificing expressive power of the grammar. Executing individual rules with more than a single thread increases utilization and is a better fit to the SIMD processing model on the GPU. By explicitly modeling rule dependence in a three stage process, we were able to provide both geometric context-sensitivity and parallelism across different rules. This approach yields the first context-sensitive parallel shape grammar for architecture, which is not only significantly faster than its CPU counterparts, but even faster than copying the data from the CPU. Using information provided during the derivation process, we built a real-time rendering pipeline for procedural cities, which is able to render cities with more than 50 000 context-sensitive buildings in real-time.

Using our parallel shape grammar, the authoring and editing of procedural cities can be enriched with instant feedback mechanisms, and both offline and real-time rendering can be significantly improved. This suggests benefits for the procedural generation of architecture in the game industry and related domains. Based on the results presented in this chapter, we can state that our scheduling system allows a wider range of algorithms to be executed on massively parallel hardware (**O6**).

Chapter 11

Conclusion

Contents

11.1 Summary

In this work we have presented a GPU programming and execution model tailored to distributed, time-varying parallelism. Based on three basic concepts (events, procedures, and work descriptors), we could implement a variety of different algorithms and execute them efficiently. The core of the presented processing model is formed by highly efficient work queues. While designing our work queues, we discovered that the best lock-free queues perform significantly worse than well-designed blocking queues, when tens of thousands of threads concurrently access the queue. The ideas of *optimistic concurrency control* many previously proposed lock-free algorithms are based upon do not hold when faced with the sheer number of threads that concurrently request resources on a GPU. Our queuing strategy, however, assigns individual queue positions to different threads and uses distributed locks to coordinate the access to these spots. With this strategy, we implemented the currently fastest queue for task management on the GPU. It serves enqueue and dequeue requests in approximately constant time, irrespective of the number of threads concurrently accessing the queue. Hence, strategies to reduce the pressure on a single queue by providing individual queues for every worker block, only offer marginal potential for speedups. Our measurements also confirmed that per-block queuing is approximately as fast as using a single queue, with the drawbacks of additional memory requirements and the need for dedicated load balancing strategies. However, keeping a small queue in fast on-chip shared memory for each worker block turned out to be a very good strategy. As shared memory is orders of magnitude faster than global memory and queue management can be performed more efficiently, locally buffered queuing was able to significantly boost performance, especially when many small items needed to be queued.

We combined our queuing strategies with a versatile megakernel approach, which overcomes most of the drawbacks of previous persistent threads and megakernel approaches. By using individual queues for each procedure, we can efficiently group items and item sets to supply work for all threads of a worker block. Exploiting low-level features of the PTX instruction set, we are able to execute multiple workpackages concurrently in the same block and supply each procedure with individual synchronization primitives. Additionally, we reuse shared memory for the execution of different procedures. In this way, our versatile megakernel can adapt to the execution of items, item sets and workpackages with arbitrary thread counts, driving GPU utilization to a maximum. Our measurements proved that our versatile megakernel greatly increases performance over previous megakernel approaches. Also, with our versatile megakernel, executing workpackages of different granularity does not decrease performance compared to the execution of equal workpackage sizes. Based on the combination of our massively parallel queues and our versatile megakernel approach, our synthetic tests showed that dynamic algorithms can be autonomously executed on the GPU with high performance, even if the individual tasks involve varying granularities of parallelism. Thus, we can state that **O1** and **O2** have been met, when presented with synthetic workloads.

While we were striving towards being more versatile and making all strategies as dynamic as possible, we also found that being too dynamic and too general can hurt performance. Our approach to provide a queue in shared memory that adapts to the presented workloads by dynamically assigning queue space to the available elements only performed well in a single test. Most often, our dynamic local queue was clearly outperformed by our static local queue, which does not suffer from additional management overhead.

While the strategy of dynamically allocating shared memory was not successful, our dynamic memory allocator for global memory proved to be very successful. Our allocator, ScatterAlloc, scatters memory allocation requests to different memory locations using a hash function that takes the requested memory size, the multiprocessor id and the thread id into account. In this way, we reduce the number of threads trying to access the same memory location, avoid retries and reduce congestion. However, when the allocator runs low on memory, the time to serve enqueue requests increases rapidly, as most of the time is spent on finding a free spot. To avoid this situation from happening, our allocator can dynamically increase the memory pool. ScatterAlloc turned out to be 100 times faster than the memory allocator provided as part of CUDA and 10 times faster than the current state of the art. ScatterAlloc is able to serve memory requests in approximately constant time, independent of the number of threads concurrently requesting memory. Thus, ScatterAlloc provides highly efficient dynamic memory management on the GPU, even if tens of thousands of threads try to allocate memory concurrently. Thus, we can state that **O3** has been met.

One noteworthy strategy we used in both, queuing and memory management, is the strategy to locally combine requests within warps. Whenever multiple threads within the same warp enqueue elements or request memory, we determine their number using warp voting mechanism and nominate a leading thread. This leader then performs the operations to global memory and communicates the results back across the warp. In this way, it is possible to speed up requests by up to one order of magnitude. At the same time, this strategy is transparent to the programmer using the memory allocator or queue.

Next to the ability to provide efficient resource allocation and keep GPU utilization high, one of the key features of our processing model is the ability to exert external influence on the execution on the GPU. We proposed two queues that offer priority scheduling on the GPU. Our priority-bucket queue can efficiently and effectively handle static priorities and react on new high priority workpackages immediately. Our priority-sorted monolithic queue devotes some processing power to keeping the work queues sorted according to individual priorities. As the queues are constantly sorted, priorities are allowed to change continuously. While the priority-bucket queue achieves scheduling accuracies proportional to the number of buckets being used, the priority-sorted queue's ability to follow priorities depends on the number of elements in the queues and the average execution time of elements. Still, both queues can provide scheduling accuracies of up to 99%. Using both queues, different scheduling policies can be implemented, including fair scheduling, time-quota-oriented scheduling, earliest job first, or earliest deadline first. As an example, we have shown that a time-quota-based scheduler can follow the target quotas very closely. With our priority queues, scheduling on the GPU becomes highly controllable and can react on changed objectives without interrupting execution. Thus, we can state that **O4** has been met.

Combining the previously mentioned components, it is possible to efficiently initiate the execution of dynamic algorithm and react on priority changes. However, execution configuration might run into suboptimal setups during execution, if a subset of threads finish early or take different execution paths. To counteract this so-called thread divergence, we propose a monitoring mechanism, which regularly checks the current state of the execution and, in case too many threads diverge, stops their execution and recombines them with other threads taking the same execution paths. Similarly to other authors who experimented with active thread compaction, we also recorded a severe overhead of thread compaction. Thus, aggressive thread compaction strategies decreased overall performance, although they strongly reduced divergence. However, with less aggressive setups, allowing divergence to a certain extent and only applying re-convergence in severe cases, we were able to boost performance in most cases. Thus, we can state that it is possible to automatically detect and counteract divergence to increase the overall performance. Thus, we can state that **O5** has also been met.

In the last three chapters of this thesis we showed various real-world applications that can profit from our scheduling model.

Our version of dynamic mesh simplification was two times faster than a state-of-the-art kernel based implementation. Detecting divergence during path tracing and correcting unhealthy setups, could increase performance by 15% . Prioritizing those image areas that yield the biggest increase in image quality, we were able to increase the convergence rate of Monte-Carlo-based global illumination methods. Building on time-aware scheduling, we present a dynamically adapting image-based rendering algorithm, which synchronizes itself to the given camera frame rates. Furthermore, we used prioritizing to focus procressing power on those image regions, which require high quality reconstruction, generating a high quality rendering while staying within the time budged. Furtheremore, we implemented a GPU-based model file parser, executing twice as fast as the GPU parser proposed by Hou *et al.* [46]. Finally, we presented a real-time Reyes pipeline, rendering 100 million micropolygons at 40 frames per second.

In the field of visualization, we have analyzed three volume rendering techniques. For all three techniques, we have found that GPU-specific issues such as thread divergence or GPU underutilization are inherent to these algorithms. Using thread re-convergence, we were able to increase the performance of particle based volume rendering by a factor of up to 13. For image plane sweep volume illumination, we drew parallelism from sampling itself and avoided synchronization with the CPU, which increased the performance of up to 26 times. For volume raycasting, we could increase the number of non-diverging warps by a factor of eight. However, we could only improve performance for certain data sets, due to the overhead associated with re-convergence. Based on the provided examples, we can state that **O1 - O5** are also met for real-world scenarios.

In the final chapter of the thesis, we proposed PGA, the first GPU shape grammar implementation that supports geometric context-sensitivity. It is not only feature complete compared to state-of-the-art CPU shape grammar implementations, but also much faster. To achieve fast rule derivation, we model rules as item sets, drawing parallelism from every rule. To avoid round trips to global memory, we use locally-buffered queuing. As many different rules need to be supported, we used the dynamic version of our local buffers. For simple datasets, as supported by other GPU implementations, our GPU-based implementation is up to 2500 times faster than a CPU-based implementation, and 100 times faster than the state-of-the-art in GPU-based grammar derivation. For the most complex data sets, we are still 10 times faster than the CPU. Additionally, we integrated the rule derivation into a real-time rendering system. Building on level-of-detail terminal nodes, frustum culling, visibility culling, and frame-to-frame coherence, we enable walkthroughs and flyovers of unlimited cities. With a visibility distance of 3500m, we can keep the geometry of 47 000 buildings up to date, while the viewer moves through the city at supersonic speed. Considering that PGA, based on our scheduling framework, is the first to support a geometric context sensitive grammar on the GPU, we can state that **O6** is also met.

11.2 Lessons Learned

GPU programming is a young discipline and a series of unexpected situations might arise when trying to go beyond what has been done before on the GPU. Especially the sheer amount of parallelism, is a concept one needs to adapt to. As tens of thousands of threads access the same resources, one can be certain that any kind of race condition will be hit during the first stress test. While, at the beginnings of GPU programming, it was a very difficult task to find such errors in an algorithm, the toolsets available to GPU programmers nowadays are much more advanced, making GPU programming more of a pleasure than pain. Still, there are many issues, one might come across. The following list, gives a small insight into the most important lessons learned.

- **Being sequential in a parallel world can be good.** Even on the GPU, there is a point when sequential execution can be better than parallel execution. This point is reached, when the entire GPU is filled with threads and generating a final result involves a reduction operation. In this case, it makes sense, to let every thread do some sequential work instead of starting more threads. This avoids parts of the reduction, while still providing enough parallelism to keep the entire device occupied.
- **Guaranteed progress is of no use when it is slow.** While, in general, only non-blocking algorithms are guaranteed to work correctly on the GPU, they might not always be the best choice when it comes to performance. As we have shown with our massively parallel queues, locks can be a lot faster. While the use of locks makes

some assumptions on the specific behavior of the underlying hardware scheduler, the achieved performance gain seems to be worth the risk. Most non-blocking algorithms act like only one thread would be accessing a shared resource and, in case this assumption turns out to be wrong, simply try again. In a massively parallel environment, this strategy is not a good idea. According to our experience, it is better to distribute the load and use fine-grained locking to protect resources where necessary.

- **A leader helps.** When accessing shared resources, individual counters or locks might become a major point of contention. Often, it is reasonable to build on a two-level approach. Combining requests locally and appointing a single thread to carry out the combined request globally. This setup can be build with warp vote functions, completely transparent to the user of an algorithm.

- **Being too dynamic can hurt.** The more general and more dynamic an algorithm tries to be, the more overhead it involves. Especially when aiming for maximum performance, this overhead can become too high. Thus, dynamic strategies should only be used if an algorithm demands it. For example, if hundreds of different rules or procedures need to be able to run in the same context, a dynamic queue in shared memory can help.

- **Templates are your friend.** One of the most important concepts in GPU programming is inlining. To help the optimizer do its job and get the mot efficient code possible, avoid costly function calls (which have been supported in CUDA since the Fermi generation). When trying to implement a general framework supporting scheduling and memory management for arbitrary algorithms, keeping things inlined can become difficult. Furthermore, one might tend towards making decisions during runtime instead of compile time to provide a simpler, more adaptable framework. However, when aiming at maximum performance, it is desirable to have all function calls inlined and move as many decisions to compile time as possible. Templates are one way to make decisions during compile time, providing a general framework that can adapt to different algorithms, and enable the compiler to inline all function calls. The downside of this strategy is an increase in compile time. The compile time of big megakernels providing inline code for tens to hundreds of procedures can grow to many minutes, strongly hindering rapid prototyping as every small change requires a complete recompile.

- **The linker is your enemy.** One strategy we followed in the first iteration of Softshell was to use the CUDA linker to reduce compile time. We were willing to accept one function call for every procedure to be more flexible in terms of compilation. While the CUDA linker is able to link functions from different CUDA files, it is not very smart when it comes to register allocation. It always assumes worst case scenarios, which means that every megakernel will use the maximum register count, even if there is not a single procedure requiring the maximum register count. This fact, especially on the newest generation of CUDA devices, which support up to 255 registers per thread, has extremely negative effects on the ability to hide memory latency.

- **Heisenberg was right.** When developing a scheduling framework for the GPU, it would be good to record what is actually happening on the device. While NVIDIA introduced a very good profiler which offers statistics about individual kernel launches, a scheduling framework would actually want to provide more information. This information should at least capture, which procedure has been executed when and where. However, measuring the execution time and especially storing information about which procedure has been executed, can introduce a severe overhead. Thus, such measurements should always be treated with care and one should be aware that it is not possible to capture all information without influencing the data itself.
- **Playing stare with your code is not the solution.** In the beginning of GPU programming there was no proper way of debugging programs. There were only two ways to understand what was happening within an algorithm. Either, run a CPU version of the exactly same algorithm and debug the program step by step on the CPU, where certain race condition would be far less likely to occur, or stare at the code and try to run the program in your own head. Luckily, times have changed. Nowadays, there are at least two additional tools to be used for debugging. First, device debuggers have been made available, which halt the entire execution and allow a step by step debugging directly on the GPU. The alternative is to print debug messages from GPU code to the console. The latter strategy can especially help to track concurrency bugs. Both features significantly increased our productivity.

11.3 Directions for Future Work

Although the advancements described in this thesis significantly increase the scheduling capabilities on the GPU and extend the range of algorithms to run on graphics processors, the feature set and comfort provided by a CPU operating system (OS) is by far not matched. In the future we want to strive towards a GPU OS. Based on our current framework, some features of a GPU OS could be reached in the foreseeable future. Amongst others, these features include the scheduling of multiple application and a GPU task manager. Our goal is to handle multiple applications in the same scheduling context. With this setup it would be possible to apply scheduling across application boundaries. One could define foreground and background applications or use a quota-based scheduling strategy. In this case, algorithms with even lower degrees of parallelism could share the GPU with other algorithms to increase GPU utilization. Based on the messaging interface already provided in Softshell, applications could be monitored and influenced from an outside source, a GPU task manager. One could inspect the execution times and memory usage of applications, change their priorities, pause their execution, or even cancel individual jobs.

Another interesting project would be implementing parts of the presented scheduling algorithms in hardware, increasing the achievable performance. If parts of the scheduling framework were build in hardware, it would also be possible to combine our strategies with the hardware scheduler. In this way, the efficiency of locks could be increased. Instead of spinning, a more reasonable back-off strategy could be implemented using something like a warp yield or even sleep operation. Exposing register file allocation to the programmer could rid the versatile megakernel of its last drawback (the increased register usage). If the register file pointer could be adjusted dynamically, the number of concurrently executing procedures could also be adjusted and thus, maximum utilization could be achieved. In a final step, completely dynamic warp grouping could be set up, eliminating the barrier synchronization from the beginning of the loop in our versatile megakernel. As soon as one warp was ready for execution, it could pull in new work items to fill the warp and start executing. If it pulls in a workpackage, it could wait for a sufficient number of other warps to get ready too and then these warps would execute the workpackage cooperatively.

In the first chapter of this work, we have proposed our processing model based on events, procedures, work items and packages. While this setup helped us in effectively writing code for our example applications, we would like to advance this basic model to a more complete state. This model would provide a more detailed specification of the CUDA functions available to the programmer and a more sophisticated memory allocation and data distribution model. We would then like to evaluate this programming model in comparison to the CUDA programming model. A suitable setup would be our basic CUDA programming course, teaching one half of the group in CUDA, while the other half would learn our new programming model. Logging the invested development time could point towards the more efficient model.

We think that an alternative scheduling strategy could evolve around memory. As memory access is most often the limiting factor for algorithms executing on the GPU, it could be beneficial to model algorithms in terms of memory dependencies. A more efficient scheduling strategy would take these dependencies into account and try to find a schedule that involve as much local data as possible, avoiding round trips to global memory or shared memory completely. This setup could lead to a pull approach to scheduling. Starting with the output, the required input data could be requested, checking the data needed for this intermediate result and so on. In the end, this scheduling strategy would only execute those parts of the algorithm that are really needed and could arrange execution to group procedures requiring the same data.

Up to now, our focus was on graphics cards only. When the work on this thesis was started, the Intel Larrabee [122], a many-core x86 architecture for visual computing, was heavily discussed. Although the Larrabee project was discontinued for graphics, a variant of the architecture without special support for graphics was recently launched in form of the Intel Xeon Phi processor. This processor features up to 61 cores and can run a maximum of 244 threads in parallel. Each core is equipped with a 16-wide SIMD unit. When compared to a GPU, the number of cores and number of concurrently running threads seem rather low. However, if all SIMD units are fully utilized, about one thousand floating point instructions can be executed concurrently. Thus, the Xeon Phi might be an interesting alternative for algorithms of a lower degree of local parallelism. In the future, we will investigate to which extent our scheduling algorithms can be useful on the Xeon Phi architecture and how a GPU-based execution compares to the Xeon Phi.

During the work on parallel shape grammars, we identified a couple of directions for future research. A very fast grammar evaluation scheme opens up new possibilities. If an entire city can be built in a couple of milliseconds, a self-learning city generator could become possible. In combination with a genetic algorithm, city layouts, building designs, or even house plans could be generated autonomously by a computer. The only requirement would be a measure to evaluate a generated design. If this function would incorporate the characteristic features of the species inhabitating the generated world, plausible cities and buildings for Sci-Fi worlds could be generated.

Another direction for future research is the combination of generation and rendering in a more general fashion. In this work we showed that data can be generated faster than streaming it from the CPU. If we had an integrated renderer, which would not require the entire data to be stored in global memory, it would even be possible to generate data for rendering locally only, avoiding the round trip to global memory completely.

Given the promising results of this thesis and the advancements made by other researches and graphics cards manufactures, we are confident that the execution on graphics processors will become more flexible, more dynamic and more accessible for a wider range of algorithms, applications, and programmers.

Bibliography

[1] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. Faster, cheaper, better – a hybridization methodology to develop linear algebra software for gpus. In *GPU Computing Gems*, vol. 2. Morgan Kaufmann, 2010. Cited on page 15.

[2] J. H. Ahn. *Memory and control organizations of stream processors*. Ph.D. thesis, Stanford University, Stanford, CA, USA, 2007. AAI3253463. Cited on page 20.

[3] T. Aila and S. Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pp. 145–149. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-603-8. doi:10.1145/1572769.1572792. Cited on pages 14 and 93.

[4] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pp. 119–129. ACM, New York, NY, USA, 1998. ISBN 0-89791-989-0. doi:10.1145/277651.277678. Cited on page 14.

[5] K. E. Batcher. Sorting networks and their applications. In *Proc. Spring Joint Computer Conference*, AFIPS '68, pp. 307–314. ACM, 1968. Cited on page 81.

[6] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pp. 18:1–18:11. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-744-8. doi:10.1145/1654059.1654078. Cited on page 15.

[7] B. Beneš, O. Šťava, R. Měch, and G. Miller. Guided Procedural Modeling. *Comp. Graph. Forum*, vol. 30, no. 2, pp. 325–334, 2011. doi:10.1111/j.1467-8659.2011.01886.x. Cited on page 164.

[8] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, vol. 35, no. 11, pp. 117–128, 2000. doi:http://doi.acm.org/10.1145/356989.357000. Cited on pages 19, 115, and 117.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proc. SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pp. 207–216. ACM, 1995. ISBN 0-89791-700-6. Cited on page 28.

[10] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999. doi:10.1145/324133.324234. Cited on page 69.

[11] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Task scheduling strategies for workflow-based applications in grids. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2 - Volume 02*, CCGRID '05, pp. 759–767. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7803-9074-1. Cited on page 14.

[12] J. Breitbart. Static gpu threads and an improved scan algorithm. In *Proceedings of the 2010 conference on Parallel processing*, Euro-Par 2010, pp. 373–380. Springer-Verlag, Berlin, Heidelberg, 2011. ISBN 978-3-642-21877-4. Cited on page 15.

[13] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, 2004. Cited on page 13.

[14] D. Cederman, B. Chatterjee, and P. Tsigas. Understanding the performance of concurrent data structures on graphics processors. In *Proceedings of the 18th international conference on Parallel Processing*, Euro-Par'12, pp. 883–894. Springer-Verlag, Berlin, Heidelberg, 2012. ISBN 978-3-642-32819-0. doi:10.1007/978-3-642-32820-6_87. Cited on page 17.

[15] D. Cederman and P. Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pp. 57–64. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2008. ISBN 978-3-905674-09-5. Cited on pages 14, 35, and 37.

[16] S. Chatterjee, M. Grossman, A. Sbirlea, and V. Sarkar. Dynamic task parallelism with a GPU work-stealing runtime system. In *Proc. Workshop on Languages and Compilers for Parallel Computing*, LCPC '11, 2011. Cited on page 14.

[17] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao. Dynamic load balancing on single- and multi-gpu systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, 2010. doi:10.1109/IPDPS.2010.5470413. Cited on pages 14 and 15.

[18] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, vol. 3, no. 2, pp. 67–78, 1971. doi:10.1145/356586.356588. Cited on page 16.

[19] R. L. Cook, L. Carpenter, and E. Catmull. The reyes image rendering architecture. *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 95–102, 1987. doi:10.1145/37402.37414. Cited on page 140.

[20] N. Corporation. *CUDA C Best Practices Guide*. 2701 San Tomas Expressway, Santa Clara 95050, USA, fourth edn., 2011. Cited on page 114.

[21] B. Csébfalvi and L. Szirmay-kalos. Monte carlo volume rendering. In *Proc. of IEEE Visualization*, pp. 449–456, 2003. Cited on pages 152 and 158.

[22] A. M. Devices. *AMD Accelerated Parallel Processing OpenCL - Programming Guide*, 2013. Cited on page 21.

[23] D. Dice and A. Garthwaite. Mostly lock-free malloc. In *ISMM '02*, pp. 163–174. ACM, New York, NY, USA, 2002. Cited on pages 19 and 117.

[24] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, vol. 8, no. 9, pp. 569–, 1965. doi:10.1145/365559.365617. Cited on page 16.

[25] H. El-Rewini, T. G. Lewis, and H. H. Ali. *Task scheduling in parallel and distributed systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994. ISBN 0-13-099235-6. Cited on page 14.

[26] S. Frey and T. Ertl. PaTraCo: A Framework Enabling the Transparent and Efficient Programming of Heterogeneous Compute Networks. In *Proc. Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV10, pp. 131–140, 2010. Cited on page 15.

[27] W. Fung and T. Aamodt. Thread block compaction for efficient simt control flow. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 25–36, 2011. doi:10.1109/HPCA.2011.5749714. Cited on page 13.

[28] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proc. IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pp. 407–420. IEEE, 2007. Cited on page 13.

[29] K. Garanzha and C. Loop. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. *Computer Graphics Forum*, vol. 29, no. 2, pp. 289–298, 2010. doi:10.1111/j.1467-8659.2009.01598.x. Cited on page 14.

[30] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pp. 43–52. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-795-7. doi:10.1145/1345206.1345215. Cited on page 17.

[31] W. Gloger. ptmalloc. http://www.malloc.de/en/, 2006. Cited on page 19.

[32] N. Greene and M. Kass. Hierarchical Z-Buffer Visibility. In *Proc. SIGGRAPH'93*, pp. 231–238, 1993. Cited on page 175.

[33] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing*, p. 14, 2012. Cited on page 15.

[34] M. Hadwiger, A. Kratz, C. Sigg, and K. Bühler. Gpu-accelerated deep shadow maps for direct volume rendering. In *Proc. 21st ACM SIGGRAPH/EUROGRAPHICS*, GH '06, pp. 49–52. ACM, 2006. ISBN 3-905673-37-1. doi:10.1145/1283900.1283908. Cited on page 152.

[35] S. Haegler, P. Wonka, S. M. Arisona, L. J. V. Gool, and P. Müller. Grammar-based Encoding of Facades. *Comp. Graph. Forum*, vol. 29, no. 4, pp. 1479–1487, 2010. Cited on page 165.

[36] D. Häggander and L. Lundberg. Optimizing dynamic memory management in a multithreaded application executing on a multiprocessor. In *ICPP '98*, pp. 262–269. IEEE, Washington, DC, USA, 1998. Cited on page 19.

[37] M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. In H. Nguyen (Editor), *GPU Gems 3*. Addison Wesley, 2007. Cited on page 113.

[38] S. Hauswiesner, M. Straka, and G. Reitmayr. Coherent image-based rendering of real-world objects. In *Proc. Symposium on Interactive 3D Graphics and Games*, I3D '11, pp. 183–190. ACM, 2011. ISBN 978-1-4503-0565-5. Cited on page 137.

[39] S. Havemann. *Generative Mesh Modeling*. PhD thesis, TU Braunschweig, 2005. Cited on page 165.

[40] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pp. 355–364. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0079-7. doi:10.1145/1810479.1810540. Cited on page 17.

[41] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, 1991. doi:10.1145/114005.102808. Cited on page 16.

[42] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pp. 522–. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 0-7695-1920-2. Cited on pages 16 and 17.

[43] J. Hoberock, V. Lu, Y. Jia, and J. Hart. Stream compaction for deferred shading. In *Proc. ACM SIGGRAPH HPG*, pp. 173–180. ACM, 2009. Cited on page 14.

[44] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In *Proceedings of the 11th international conference on Principles of distributed systems*, OPODIS'07, pp. 401–414. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 3-540-77095-X, 978-3-540-77095-4. Cited on pages 17 and 52.

[45] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: portable stream programming on graphics engines. In *Proc. Architectural support for programming languages and operating systems*, ASPLOS '11, pp. 381–392. ACM, 2011. Cited on page 15.

[46] Q. Hou, K. Zhou, and B. Guo. BSGP: bulk-synchronous GPU programming. *ACM Trans. Graph.*, vol. 27, no. 3, pp. 19:1–19:12, 2008. Cited on pages 15, 139, and 187.

[47] Q. Hou, K. Zhou, and B. Guo. Debugging GPU stream programs through automatic dataflow recording and visualization. *ACM Trans. Graph.*, vol. 28, no. 5, pp. 153:1–153:11, 2009. Cited on page 32.

[48] X. Huang, C. Rodrigues, S. Jones, I. Buck, and W. mei Hwu. XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines. In *Computer and Information Technology (CIT), 2010 IEEE*, pp. 1134 –1139, 2010. Cited on pages 20, 114, and 126.

[49] R. L. Hudson, B. Saha, A. Adl-Tabatabai, and B. C. Hertzberg. McRT-Malloc: a scalable transactional memory allocator. In *ISMM '06*, pp. 74–83. ACM, New York, NY, USA, 2006. Cited on page 19.

[50] X. Huo, S. Krishnamoorthy, and G. Agrawal. Efficient scheduling of recursive control flow on gpus. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pp. 409–420. ACM, New York, NY, USA, 2013. ISBN 978-1-4503-2130-3. doi:10.1145/2464996.2479870. Cited on page 15.

[51] R. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, vol. 2, no. 1, 1973. Cited on page 175.

[52] B. Kainz, M. Grabner, A. Bornik, S. Hauswiesner, J. Muehl, and D. Schmalstieg. Ray casting of multiple volumetric datasets with polyhedral boundaries on manycore gpus. *ACM Trans. Graph.*, vol. 28, no. 5, pp. 1–9, 2009. doi:http://doi.acm.org/10.1145/1618452.1618498. Cited on page 152.

[53] J. T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, vol. 20, no. 4, pp. 143–150, 1986. Cited on page 134.

[54] J. Kalojanov, M. Billeter, and P. Slusallek. Two-level grids for ray tracing on gpus. In *Computer Graphics Forum*, vol. 30, pp. 307–314, 2011. Cited on page 14.

[55] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. USENIX annual technical conference*, USENIXATC'11, pp. 2–2. USENIX Association, 2011. Cited on page 16.

[56] Khronos. *OpenCL The standard for heterogeneous parallel programming*. Khronos OpenCL Working Group, 2008. Cited on pages 2 and 13.

[57] J. Kim. *Efficient rendering of large 3-D and 4-D scalar fields*. Ph.D. thesis, University of Maryland, 2008. Cited on page 152.

[58] K. C. Knowlton. A Fast Storage Allocator. *Commun. ACM*, vol. 8, no. 10, pp. 623–624, 1965. doi:10.1145/365628.365655. Cited on page 177.

[59] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973. Cited on page 120.

[60] L. Krecklau, J. Born, and L. Kobbelt. View-Dependent Realtime Rendering of Procedural Facades with High Geometric Detail . *Comp. Graph. Forum*, vol. 32, no. 2pt1, 2013. Cited on page 165.

[61] L. Krecklau and L. Kobbelt. Procedural Modeling of Interconnected Structures. *Comp. Graph. Forum*, vol. 30, p. 335ff., 2011. Cited on page 164.

[62] L. Krecklau, D. Pavic, and L. Kobbelt. Generalized Use of Non-Terminal Symbols for Procedural Modeling. *Comp. Graph. Forum*, vol. 29, pp. 2291–2303, 2011. Cited on page 164.

[63] T. Kroes, F. H. Post, and C. P. Botha. Exposure render: An interactive photo-realistic volume rendering framework. *PLoS ONE*, vol. 7, no. 7, p. e38586, 2012. doi:10.1371/journal.pone.0038586. Cited on page 152.

[64] J. Krüger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *Proc. IEEE Vis.*, pp. 287 –292, 2003. doi:10.1109/VISUAL.2003.1250384. Cited on page 152.

[65] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, 1981. doi:10.1145/319566.319567. Cited on page 17.

[66] P. Lacz and J. Hart. Procedural Geometry Synthesis on the GPU. In *Workshop on General Purpose Computing on Graphics Processors*, pp. 23–23, 2004. Cited on pages 165 and 170.

[67] S. Laine, T. Karras, and T. Aila. Megakernels considered harmful: Wavefront path tracing on gpus. In *Proc. High-Performance Graphics*, 2013. Cited on pages 23, 94, and 95.

[68] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 2, pp. 190–222, 1983. doi:10.1145/69624.357207. Cited on page 17.

[69] P.-A. Larson and M. Krishnan. Memory allocation for long-running server applications. *SIGPLAN Not.*, vol. 34, pp. 176–185, 1998. Cited on page 120.

[70] C. Lauterbach, Q. Mo, and D. Manocha. gproximity: Hierarchical gpu-based operations for collision and distance queries. *Comput. Graph. Forum*, vol. 29, no. 2, pp. 419–428, 2010. Cited on page 20.

[71] D. Lea. A memory allocator, 1996. Unix/Mail, http://gee.cs.oswego.edu/dl/html/malloc.html. Cited on pages 19 and 117.

[72] P. P. C. Lee, T. Bu, and G. Chandranmenon. A lock-free, cache-efficient shared ring buffer for multi-core architectures. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '09, pp. 78–79. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-630-4. doi:10.1145/1882486.1882508. Cited on page 17.

[73] S. Lefebvre, S. Hornus, and A. Lasram. By-example synthesis of architectural textures. *ACM Trans. Graph.*, vol. 29, p. A84, 2010. Cited on page 165.

[74] M. Levoy. Efficient ray tracing of volume data. *ACM Trans. Graph.*, vol. 9, no. 3, pp. 245–261, 1990. Cited on pages 151 and 152.

[75] Y. Li, F. Bao, E. Zhang, Y. Kobayashi, and P. Wonka. Geometry Synthesis on Surfaces Using Field-Guided Shape Grammars. *IEEE Trans. Visualization and Computer Graphics*, vol. 17, no. 2, pp. 231–243, 2011. Cited on page 164.

[76] J. Lin, D. Cohen-Or, H. Zhang, C. Liang, A. Sharf, O. Deussen, and B. Chen. Structure-preserving retargeting of irregular 3D architecture. *ACM Trans. Graph.*, vol. 30, no. 6, p. A183, 2011. Cited on page 165.

[77] F. Lindemann and T. Ropinski. About the influence of illumination models on image comprehension in direct volume rendering. *IEEE TVCG*, vol. 17, no. 12, pp. 1922 –1931, 2011. doi:10.1109/TVCG.2011.161. Cited on page 152.

[78] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA tesla: A unified graphics and computing architecture. *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, 2008. doi:10.1109/MM.2008.31. Cited on pages 20 and 114.

[79] M. Lipp, P. Wonka, and M. Wimmer. Parallel Generation of Multiple L-systems. *Computer and Graphics*, vol. 34, no. 5, 2010. Cited on pages 165 and 166.

[80] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proc. SIGGRAPH '97*, pp. 199–208. ACM, 1997. ISBN 0-89791-896-7. Cited on pages 24, 26, and 133.

[81] M. Magdics. Real-time Generation of L-system Scene Models for Rendering and Interaction. In *Spring Conf. on Computer Graphics*, pp. 77–84. Comenius Univ., 2009. Cited on pages 165 and 166.

[82] L. Marsalek, A. Hauber, and P. Slusallek. High-speed volume ray casting with cuda. In *Proc. IEEE Interactive Ray Tracing*, pp. 185–185. IEEE, 2008. Cited on page 152.

[83] J.-E. Marvie, C. Buron, P. Gautron, P. Hirtzlin, and G. Sourimant. GPU Shape Grammars. *Comp. Graph. Forum*, vol. 31, no. 7-1, pp. 2087–2095, 2012. Cited on pages 165, 166, and 179.

[84] J.-E. Marvie, G. Pascal, P. Hirtzlin, and S. Gael. Render-Time Procedural Per-Pixel Geometry Generation. In *Graphics Interface*, pp. 167–174, 2011. Cited on page 165.

[85] H. Massalin and C. Pu. A lock-free multiprocessor os kernel. *SIGOPS Oper. Syst. Rev.*, vol. 26, no. 2, pp. 8–, 1992. Cited on page 16.

[86] W. Matusik, C. Buehler, R. Raskar, S. J. Gortler, and L. McMillan. Image-based visual hulls. In *Proc. SIGGRAPH '00*, SIGGRAPH '00, pp. 369–374. ACM, 2000. ISBN 1-58113-208-5. Cited on page 136.

[87] A. Maximo, R. Marroquim, and R. Farias. Hardware-assisted projected tetrahedra. In *Proc. EuroVis*, EuroVis'10, pp. 903–912. Eurographics Association, 2010. doi:10.1111/j.1467-8659.2009.01673.x. Cited on page 152.

[88] M. D. McCool, Z. Qin, and T. S. Popa. Shader metaprogramming. In *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, pp. 57–68, 2002. ISBN 1-58113-580-7. Cited on page 13.

[89] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pp. 235–246. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0053-7. doi:10.1145/1815961.1815992. Cited on page 13.

[90] P. Merrell and D. Manocha. Model Synthesis: A General Procedural Modeling Algorithm. *Visualization and Computer Graphics, IEEE Trans.*, vol. 17, no. 6, pp. 715–728, 2011. Cited on page 165.

[91] M. M. Michael. Scalable lock-free dynamic memory allocation. *SIGPLAN Not.*, vol. 39, no. 6, pp. 35–46, 2004. doi:http://doi.acm.org/10.1145/996893.996848. Cited on pages 19 and 117.

[92] M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Tech. rep., Rochester, NY, USA, 1995. Cited on page 17.

[93] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pp. 267–275. ACM, New York, NY, USA, 1996. ISBN 0-89791-800-2. doi:10.1145/248052.248106. Cited on pages 17 and 52.

[94] D. Molka, D. Hackenberg, R. Schone, and M. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09.*, pp. 261 –270, 2009. Cited on page 114.

[95] K. Mueller and R. Yagel. Fast perspective volume rendering with splatting by utilizing a ray-driven approach. In *Proc. VIS*, VIS '96, pp. 65–ff. IEEE, 1996. ISBN 0-89791-864-9. Cited on page 152.

[96] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. V. Gool. Procedural Modeling of Buildings. *ACM Trans. Graph.*, vol. 25, no. 3, pp. 614–623, 2006. doi:10.1145/1179352.1141931. Cited on pages 164 and 166.

[97] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pp. 308–317. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-1053-6. doi:10.1145/2155620.2155656. Cited on page 13.

[98] D. M. Nicol. Inflated speedups in parallel simulations via malloc(). *International Journal on Simulation*, vol. 2, pp. 413–426, 1992. Cited on pages 19 and 113.

[99] J. Novák, V. Havran, and C. Dachsbacher. Path regeneration for interactive path tracing. *EG Short papers*, pp. 61–64, 2010. Cited on page 14.

[100] NVIDIA. NVIDIA's next generation CUDA compute architecture: Fermi. White paper, 2009. Available online. Cited on pages 13, 21, and 113.

[101] NVIDIA. *CUDA Dynamic Parallelism Programming Guide*. NVIDIA, 2012. Cited on page 15.

[102] NVIDIA. NVIDIA's next generation CUDA compute architecture: Kepler tm gk110. White paper, 2012. Available online. Cited on page 13.

[103] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*. NVIDIA, 2013. Cited on pages 2, 13, and 113.

[104] Y. I. H. Parish and P. Müller. Procedural modeling of cities. In *Proc. SIGGRAPH '01*, pp. 301–308, 2001. doi:10.1145/383259.383292. Cited on page 164.

[105] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. Optix: a general purpose ray tracing engine. *ACM Trans. Graph.*, vol. 29, no. 4, pp. 66:1–66:13, 2010. doi:10.1145/1778765.1778803. Cited on pages 15, 32, 94, 100, and 134.

[106] A. Patney and J. D. Owens. Real-time reyes-style adaptive surface subdivision. *ACM Trans. Graph.*, vol. 27, no. 5, pp. 143:1–143:8, 2008. doi:10.1145/1409060.1409096. Cited on page 140.

[107] G. Patow and G. Besuievsky. Challenges in Procedural Modeling of Buildings. In *Eurographics Workshop on Urban Data Modelling and Visualisation*, 2013. Cited on page 163.

[108] H. Poincaré. *The Measure of Time*. New York: Science Press, 1913. Cited on page 33.

[109] P. Prusinkiewicz, M. James, and R. Měch. Synthetic Topiary. In *Proc. SIGGRAPH 94*, pp. 351–358, 1994. doi:10.1145/192161.192254. Cited on page 164.

[110] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, 1990. ISBN 0-387-97297-8. Cited on page 164.

[111] P. Prusinkiewicz, L. Mündermann, R. Karwowski, and B. Lane. The Use of Positional Information in the Modeling of Plants. In *Proc. SIGGRAPH 2001*, pp. 289–300, 2001. doi:10.1145/383259.383291. Cited on page 164.

[112] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, vol. 33, pp. 668–676, 1990. Cited on page 117.

[113] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage-rasterization. In *Proc. EG/SIGGRAPH Graphics Hardware*, pp. 109–118, 2000. Cited on page 152.

[114] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart hardware-accelerated volume rendering. In *Proc. of Data visualisation*, pp. 231–238. Eurographics Association, 2003. Cited on page 152.

[115] T. Ropinski, C. Doring, and C. Rezk-Salama. Interactive volumetric lighting simulating scattering and shadowing. In *Proc. IEEE PacificVis*, pp. 169 –176, 2010. doi:10.1109/PACIFICVIS.2010.5429594. Cited on page 152.

[116] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: operating system abstractions to manage gpus as compute devices. In *Proc. ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 233–248. ACM, 2011. Cited on page 15.

[117] N. Sakamoto, J. Nonaka, K. Koyamada, and S. Tanaka. Particle-based volume rendering. In *Visualization, 2007. APVIS*, pp. 129 –132, 2007. doi:10.1109/APVIS.2007.329287. Cited on pages 151, 152, and 158.

[118] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis. Dynamic fine-grain scheduling of pipeline parallelism. In *Proc. International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pp. 22–32. IEEE, 2011. Cited on page 15.

[119] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pp. 1–10. IEEE Computer Society, Washington, DC, USA, 2009. ISBN 978-1-4244-3751-1. doi:10.1109/IPDPS.2009.5161005. Cited on page 15.

[120] P. Schlegel and R. Pajarola. Layered volume splatting. In *Proc. ISVC*, ISVC '09, pp. 1–12. Berlin, Heidelberg, 2009. ISBN 978-3-642-10519-7. doi:10.1007/978-3-642-10520-3_1. Cited on page 152.

[121] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *ISMM '06*, pp. 84–94. ACM, New York, NY, USA, 2006. ISBN 1-59593-221-6. Cited on pages 19 and 117.

[122] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, vol. 27, no. 3, pp. 18:1–18:15, 2008. Cited on pages 21 and 191.

[123] S. Seo, J. Kim, and J. Lee. Sfmalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores. In *PACT '11*, pp. 253–263. IEEE Computer Society, Washington, DC, USA, 2011. Cited on pages 19 and 117.

[124] L. Sha, T. Abdelzaher, K.-E. Arzen, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Syst.*, vol. 28, no. 2, pp. 101–155, 2004. Cited on page 16.

[125] C.-H. Shann, T.-L. Huang, and C. Chen. A practical nonblocking queue algorithm using compare-and-swap. In *Parallel and Distributed Systems, 2000. Proceedings. Seventh International Conference on*, pp. 470–475, 2000. doi:10.1109/ICPADS.2000.857731. Cited on pages 18 and 52.

[126] K. Shirahata, H. Sato, and S. Matsuoka. Hybrid map task scheduling for gpu-based heterogeneous clusters. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pp. 733–740. IEEE Computer Society, Washington, DC, USA, 2010. ISBN 978-0-7695-4302-4. doi:10.1109/CloudCom.2010.55. Cited on page 15.

[127] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. In *Proc. ACVM Volume Visualization*, VVS '90, pp. 63–70. ACM, 1990. ISBN 0-89791-417-1. doi:10.1145/99307.99322. Cited on page 152.

[128] V. Šoltészová, D. Patel, and I. Viola. Chromatic shadows for improved perception. In *Proc. SIGGRAPH NPAR*, NPAR '11, pp. 105–116. ACM, 2011. ISBN 978-1-4503-0907-3. doi:10.1145/2024676.2024694. Cited on page 152.

[129] J. Stankovic, K. Ramamritham, and S. Cheng. Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems. *Computers, IEEE Transactions on*, vol. C-34, no. 12, pp. 1130–1143, 1985. doi:10.1109/TC.1985.6312211. Cited on page 14.

[130] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Proc. IEEE Volume Graphics*, VG'05, pp. 187–195. Eurographics Association, 2005. ISBN 3-905673-26-6. doi:10.2312/VG/VG05/187-195. Cited on page 152.

[131] R. Stephens. A survey of stream processing, 1995. Cited on page 22.

[132] G. Stiny. *Pictorial and Formal Aspects of Shape and Shape Grammars*. Birkhauser Verlag, Basel, 1975. Cited on page 164.

[133] G. Stiny. Spatial Relations and Grammars. *Environment and Planning B*, vol. 9, pp. 313–314, 1982. Cited on page 164.

[134] J. A. Stuart and J. D. Owens. Message passing on data-parallel architectures. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pp. 1–12. IEEE Computer Society, Washington, DC, USA, 2009. ISBN 978-1-4244-3751-1. doi:10.1109/IPDPS.2009.5161065. Cited on page 15.

[135] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan. GRAMPS: A programming model for graphics pipelines. *ACM Trans. Graph.*, vol. 28, no. 1, pp. 1–11, 2009. Cited on pages 15 and 21.

[136] E. Sundén, A. Ynnerman, and T. Ropinski. Image Plane Sweep Volume Illumination. *IEEE TVCG(Vis Proceedings)*, vol. 17, no. 12, pp. 2125–2134, 2011. Cited on pages 151, 152, and 154.

[137] K.-C. Tai. Definitions and detection of deadlock, livelock, and starvation in concurrent programs. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 02*, ICPP '94, pp. 69–72. IEEE Computer Society, Washington, DC, USA, 1994. ISBN 0-8493-2493-9. doi:10.1109/ICPP.1994.84. Cited on page 16.

[138] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007. ISBN 9780136006633. Cited on page 13.

[139] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 3, pp. 260–274, 2002. doi:10.1109/71.993206. Cited on page 15.

[140] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '01, pp. 134–143. ACM, New York, NY, USA, 2001. ISBN 1-58113-409-6. doi:10.1145/378580.378611. Cited on pages 18 and 52.

[141] S. Tzeng, A. Patney, and J. D. Owens. Task management for irregular-parallel workloads on the gpu. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pp. 29–37. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2010. Cited on pages 14, 37, 69, and 140.

[142] J. D. Valois. Implementing lock-free queues. In *In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV*, pp. 64–69, 1994. Cited on pages 17 and 44.

[143] D. van Antwerpen. Improving simd efficiency for parallel monte carlo light transport on the gpu. In *Proc. ACM SIGGRAPH HPG*, pp. 41–50. ACM, 2011. Cited on page 14.

[144] I. Wald. Active thread compaction for gpu path tracing. In *Proc. ACM SIGGRAPH HPG*, pp. 51–58. ACM, 2011. Cited on page 14.

[145] L. A. Westover. Splatting: A parallel, feed-forward volume rendering algorithm. Tech. rep., 1991. Cited on page 152.

[146] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*, 1995. Cited on page 19.

[147] P. Wonka, M. Wimmer, F. X. Sillion, and W. Ribarsky. Instant Architecture. *ACM Trans. Graph.*, vol. 22, no. 3, pp. 669–677, 2003. doi:10.1145/1201775.882324. Cited on pages 164 and 165.

[148] S. Xiao and W. chun Feng. Inter-block gpu communication via fast barrier synchronization. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, 2010. doi:10.1109/IPDPS.2010.5470477. Cited on page 15.

[149] Z. Xu, X. Hou, and J. Sun. Ant algorithm-based task scheduling in grid computing. In *Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference on*, vol. 2, pp. 1107–1110 vol.2, 2003. doi:10.1109/CCECE.2003.1226090. Cited on page 14.

[150] S. Yan, G. Long, and Y. Zhang. Streamscan: fast scan algorithms for gpus without global barrier synchronization. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pp. 229–238. ACM, New York, NY, USA, 2013. ISBN 978-1-4503-1922-5. doi:10.1145/2442516.2442539. Cited on page 15.

[151] T. Yang and A. Gerasoulis. Pyrros: static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the 6th international conference on Supercomputing*, ICS '92, pp. 428–437. ACM, New York, NY, USA, 1992. ISBN 0-89791-485-6. doi:10.1145/143369.143446. Cited on page 14.

[152] T. Yang, Z. Huang, X. Lin, J. Chen, and J. Ni. A Parallel Algorithm for Binary-Tree-Based String Rewriting in L-system. In *Proc. of the Second International Multi-symposiums of Computer and Computational Sciences*, pp. 245–252, 2007. Cited on page 165.

[153] K. Zhou, M. Gong, X. Huang, and B. Guo. Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 5, pp. 669–681, 2011. Cited on pages 25 and 133.

[154] K. Zhou, Q. Hou, Z. Ren, M. Gong, X. Sun, and B. Guo. RenderAnts: interactive Reyes rendering on GPUs. *ACM Trans. Graph.*, vol. 28, no. 5, pp. 155:1–155:11, 2009. Cited on pages 15 and 140.

[155] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, vol. 27, pp. 126:1–126:11, 2008. Cited on page 20.

Appendix A

Softshell Source Code Examples

A.1 Mesh Simplification

This sections shows the source code of the comparison implementation between Softshell and CUDA for the mesh simplification testcase.

Listing A.1: Common Header

```


#pragma once

#include <memory>
#include <cutil_math.h>
#include "GLBuffer.h"


#define NO_ACTIVE_ENTRY 0xFFFFFFFF

struct OctreeEntry
{
  unsigned int vertexOffset;
  unsigned int numVertices;
  unsigned int triangleOffset;
  unsigned int numTriangles;

  #define NO_CHILD_FLAG 0xFFFFFFFF
  #define MAX_DEPTH 32
  __host__ __device__ static  inline bool hasChildren(unsigned
      entry)
  {
    return entry != 0xFFFFFFFF;
  }
  unsigned int children; //(ccciiiiiiiiiiiiiiiiiiiiiiiiiiiii)
                          //c is count of children - 1
                          //i index of first child
  #define CHILD_ID_SHIFT 0
  #define CHILD_COUNT_SHIFT 29
  #define CHILD_ID_BITS 0x1FFFFFFF
  #define CHILD_ID_MASK CHILD_ID_BITS
  #define CHILD_COUNT_MASK 0xE0000000
  #define CHILD_COUNT_BITS 0x7
  __host__ __device__ static inline unsigned getNumChildren(unsigned entry)
  {
    return ((entry >> CHILD_COUNT_SHIFT) & CHILD_COUNT_BITS)+1;
  }
  __host__ __device__ static inline unsigned setNumChildren(unsigned num, unsigned entry)
  {
    return (((num-1) & CHILD_COUNT_BITS) << CHILD_COUNT_SHIFT) | (entry & CHILD_ID_BITS);
  }
  __host__ __device__ static inline unsigned getFirstChild(unsigned entry)
  {
    return (entry >> CHILD_ID_SHIFT) & CHILD_ID_BITS;
  }
  __host__ __device__ static inline unsigned setFirstChild(unsigned child, unsigned entry)
  {
    return (entry & CHILD_COUNT_MASK) | ((child & CHILD_ID_BITS) << CHILD_ID_SHIFT);
  }
  OctreeEntry()
  {
    vertexOffset = numVertices = triangleOffset = numTriangles = 0;
    children = NO_CHILD_FLAG;
  }

};


struct CUDAStorages
{
  GL::Buffer vertex_buffer;
  GL::Buffer normal_buffer;
  GL::Buffer index_buffer;


  cuda_ptr<OctreeEntry> d_octree_data;
  cuda_ptr<unsigned int> d_octree_vertex_coherence;
  cuda_ptr<unsigned int> d_octree_triangle_coherence;
  cuda_ptr<unsigned int> d_vertex_lookup;

  size_t active_set;
  unsigned int active_nodes_count;
  cuda_ptr<unsigned int> d_active_set[2];


  size_t vertexcount;
  size_t trianglecount;
  size_t voxelcount;
  size_t leafs;
  size_t maxdepth;

  std::unique_ptr<unsigned int[]> h_levelOffset;
  cuda_ptr<uint3> d_voxel_ids;

  size_t reductionClusterSize;

  size_t triangle_coherence_size;
  size_t vertex_coherence_size;

  float boundaries[6];

  CUDAStorages();
  ~CUDAStorages();
};
```

Listing A.2: Kernel-Based Implementation

```


  __constant__ OctreeEntry* octree;
  __constant__ unsigned int *vertexIdOut, *triOutput;
  __constant__ const unsigned int *vertexCoherence, *
      triangleCoherence;
  __constant__ unsigned int vertices, maxdepth;
  __constant__ const uint3* voxelIds;


  __device__ uint mark(uint node)
  {
    return 0x80000000 | node;
  }
  __device__ uint unmark(uint node)
  {
    return (~0x80000000) & node;
  }
  __device__ bool isMarked(uint node)
  {
    return ((node & 0x80000000) == 0x80000000)?true:false;
  }

  template<bool DEPTHERROR>
  __global__ void checkLevel(uint* activeSet, uint*
      numberofchildren, uint* numberOfLeafs, uint depth, uint num,
      float threshold)
  {
    uint id = blockDim.x * blockIdx.x + threadIdx.x;
    if(id >= num) return;
    uint mynodeid = activeSet[id];
    uint childnodes = 0, leafnodes = 0;

    const OctreeEntry* node = octree + mynodeid;
    uint numChildren = OctreeEntry::hasChildren(node->children)?
       OctreeEntry::getNumChildren(node->children):0;
    uint firstChildId = OctreeEntry::getFirstChild(node->children)
       ;
    for(uint i = 0; i < numChildren; ++i)
    {
      uint thisChildId = firstChildId + i;
      const OctreeEntry *childEntry = octree + thisChildId;
      //get voxel boundaries
      float3 vMin, vMax;
      getVoxelBounds(voxelIds[thisChildId], depth+1, vMin, vMax);
      //check if criterion is fullfilled
      bool expand = carryOnTraversal<DEPTHERROR>(depth+1, vMin,
         vMax, threshold);
      if(expand && (depth+1) < maxdepth)
        ++childnodes;
      else //leaf
        ++leafnodes;
    }
    if(numChildren == 0)
       ++leafnodes;

    numberofchildren[id] = childnodes;
    numberOfLeafs[id] = leafnodes;
  }

template<bool DEPTHERROR>
 __global__ void writeLevel(const uint* activeSet, const uint*
     numberofchildren, const  uint* numberOfLeafs, uint* children,
      uint* childTriangles, uint* leafs, uint depth, uint num,
     float threshold)
  {
    uint id = blockDim.x * blockIdx.x + threadIdx.x;
    if(id >= num) return;
    uint mynodeid = activeSet[id];
    uint childnodes_offset = numberofchildren[id];
    uint leafnodes_offset = numberOfLeafs[id];

    const OctreeEntry* node = octree + mynodeid;
    uint numChildren = OctreeEntry::hasChildren(node->children)?
       OctreeEntry::getNumChildren(node->children):0;
    uint firstChildId = OctreeEntry::getFirstChild(node->children)
       ;
    for(uint i = 0; i < numChildren; ++i)
    {
      uint thisChildId = firstChildId + i;
      const OctreeEntry *childEntry = octree + thisChildId;
      //get voxel boundaries
    float3 vMin, vMax;
    getVoxelBounds(voxelIds[thisChildId], depth+1, vMin, vMax);
    //check if criterion is fullfilled
    bool expand = carryOnTraversal<DEPTHERROR>(depth+1, vMin, vMax, threshold);
    if(expand && (depth+1) < maxdepth)
    {
      children[childnodes_offset] = thisChildId;
      childTriangles[childnodes_offset++] = childEntry->numTriangles;
    }
    else //leaf
      leafs[leafnodes_offset++] = thisChildId;
  }
  if(numChildren == 0)
     leafs[leafnodes_offset++] = (depth + 1 >= maxdepth)?mark(mynodeid):mynodeid;
}

__global__ void handleLeafs(const uint* leafs,  uint num)
{
  uint id = blockDim.x * blockIdx.x + threadIdx.x;
  if(id >= num) return;
  uint nodeId = leafs[id];
  bool marked = isMarked(nodeId);
  nodeId = unmark(nodeId);
  const OctreeEntry *node = octree + nodeId;

  uint vertexTargetStart = node->vertexOffset;
  uint vertexTargetEnd = vertexTargetStart + node->numVertices;
  for(uint targetVertex = vertexTargetStart; targetVertex < vertexTargetEnd; ++targetVertex)
  {
    uint coherenceId = vertexCoherence[targetVertex];
    vertexIdOut[coherenceId] = marked?coherenceId:(vertices + nodeId);
  }
}

__global__ void handleNodes(const uint* nodes, const uint* triangleOffset, uint num)
{
  uint id = blockDim.x * blockIdx.x + threadIdx.x;
  if(id >= num) return;
  uint nodeId = nodes[id];
  uint triOffset = triangleOffset[id];
  const OctreeEntry *node = octree + nodeId;

  uint numTris = node->numTriangles;
  if(numTris == 0)
    return;

    uint startWrite = triOffset;
    uint * indexoutput = triOutput + 3*startWrite;


    //run through all triangles
    for(uint i = triOffset; i < triOffset + numTris; ++i)
    {
      //all 3 vertices
      for(unsigned int v = 0; v < 3; ++v)
        *indexoutput++ = vertexIdOut[triangleCoherence[3*i + v]];
    }
  }
```

Listing A.3: Kernel-Based Control

```

#include "tools/utils.h"

#include "clustering.h"
#include "timing.h"

#include <vector>
#include "cudpp.h"

namespace CUDA
{
  Timing timing;

  #include "helpers.cuh"
  #include "cuda_kernel.cuh"

  const size_t BLOCKSIZE = 256;

  template<class T>
  T divup(T a, T b) { return (a + b - 1)/b; }

  uint *d_activeset, *d_activeset_offset, *d_leaves, *
      d_leaves_offset, *d_triangle_offset;
  CUDPPHandle theCudpp;
  CUDPPHandle scanplan;

  void prepareData(CUDAStorages* memory)
  {
     setOctreeBoundaries(memory->boundaries);
     setLevelOffsets(memory->h_levelOffset.get());

     CUDA_CHECKED_CALL(cudaMemcpyToSymbol(octree, &memory->
         d_octree_data, sizeof(OctreeEntry*)));
```

```
    CUDA_CHECKED_CALL(cudaMemcpyToSymbol(vertexCoherence, &memory
        ->d_octree_vertex_coherence, sizeof(const unsigned int*)))
        ;
    CUDA_CHECKED_CALL(cudaMemcpyToSymbol(vertexIdOut, &memory->
        d_vertex_lookup, sizeof(unsigned int *)));
    CUDA_CHECKED_CALL(cudaMemcpyToSymbol(vertices, &memory->
        vertexcount, sizeof(unsigned int)));
    CUDA_CHECKED_CALL(cudaMemcpyToSymbol(maxdepth, &memory->
        maxdepth, sizeof(unsigned int)));
    CUDA_CHECKED_CALL(cudaMemcpyToSymbol(voxelIds, &memory->
        d_voxel_ids, sizeof(const uint3*)));
    CUDA_CHECKED_CALL(cudaMemcpyToSymbol(triangleCoherence, &
        memory->d_octree_triangle_coherence, sizeof(const unsigned
         int*)));

    //data arrays
    CUDA_CHECKED_CALL(cudaMalloc(&d_activeset, sizeof(uint)*
        memory->voxelcount));
    CUDA_CHECKED_CALL(cudaMalloc(&d_activeset_offset, sizeof(uint
        )*(memory->voxelcount+1)));
    CUDA_CHECKED_CALL(cudaMalloc(&d_leaves, sizeof(uint)*(memory
        ->voxelcount)));
    CUDA_CHECKED_CALL(cudaMalloc(&d_leaves_offset, sizeof(uint)*(
        memory->voxelcount+1)));
    CUDA_CHECKED_CALL(cudaMalloc(&d_triangle_offset, sizeof(uint)
        *(memory->trianglecount)));

    cudppCreate(&theCudpp);
    CUDPPConfiguration config;
    config.op = CUDPP_ADD;
    config.datatype = CUDPP_UINT;
    config.algorithm = CUDPP_SCAN;
    config.options = CUDPP_OPTION_FORWARD |
        CUDPP_OPTION_EXCLUSIVE;
    size_t scanSize = std::max<size_t>(memory->voxelcount+1,
        memory->trianglecount);
    CUDPPResult res = cudppPlan(theCudpp, &scanplan, config,
        scanSize, 1, 0);
    if (CUDPP_SUCCESS != res)
      std::cout << "Error creating CUDPPPlan\n";

}


void cleanUp()
{
    CUDA_CHECKED_CALL(cudaFree(d_activeset));
    CUDA_CHECKED_CALL(cudaFree(d_activeset_offset));
    CUDA_CHECKED_CALL(cudaFree(d_leaves));
    CUDA_CHECKED_CALL(cudaFree(d_leaves_offset));
    CUDA_CHECKED_CALL(cudaFree(d_triangle_offset));
    cudppDestroyPlan(scanplan);
    cudppDestroy(theCudpp);
}

size_t runPrefixSum(uint* d_input, uint num)
{
  uint numEnd0, numEnd1;
  CUDA_CHECKED_CALL(cudaMemcpy(&numEnd0, d_input+(num-1), sizeof(uint), cudaMemcpyDeviceToHost));
  CUDPPResult res = cudppScan(scanplan, d_input, d_input, num);
  if (CUDPP_SUCCESS != res)
      std::cout << "Error running scan\n";
  CUDA_CHECKED_CALL(cudaMemcpy(&numEnd1, d_input+(num-1), sizeof(uint), cudaMemcpyDeviceToHost));
  return numEnd0 + numEnd1;
}

template<bool DEPTHERROR>
size_t buildMesh(CUDAStorages* memory, float threshold);

size_t buildMesh(CUDAStorages* memory, float threshold, bool error_depth)
{
  if(error_depth)
    return buildMesh<true>(memory, threshold);
  else
    return buildMesh<false>(memory, threshold);
}

template<bool DEPTHERROR>
size_t buildMesh(CUDAStorages* memory, float threshold)
{
  CUDA::GL::mapped_buffer<unsigned int> d_indices(memory->cuda_index_buffer);
  CUDA_CHECKED_CALL(cudaMemcpyToSymbol(triOutput, &d_indices.get(), sizeof(uint*)));

  size_t numnodes = 0;
  size_t numleafs = 0;
  size_t activenodes = 1;
  size_t cdepth = 0;

  timing.begin();

  uint null = 0;
  CUDA_CHECKED_CALL(cudaMemcpy(d_activeset, &null, sizeof(uint), cudaMemcpyHostToDevice));
  CUDA_CHECKED_CALL(cudaMemcpy(d_triangle_offset, &null, sizeof(uint), cudaMemcpyHostToDevice));

  while(activenodes > 0)
```

```
    {
      checkLevel<DEPTHERROR><<<divup(activenodes,BLOCKSIZE),
         BLOCKSIZE>>>(d_activeset + numnodes, d_activeset_offset,
         d_leaves_offset, cdepth, activenodes, threshold);
      size_t newNodes = runPrefixSum(d_activeset_offset,
         activenodes);
      size_t newLeafs = runPrefixSum(d_leaves_offset, activenodes)
         ;
      writeLevel<DEPTHERROR><<<divup(activenodes,BLOCKSIZE),
         BLOCKSIZE>>>(d_activeset + numnodes, d_activeset_offset,
         d_leaves_offset, d_activeset + numnodes + activenodes,
         d_triangle_offset + numnodes + activenodes, d_leaves +
         numleafs, cdepth, activenodes, threshold);
      //write vertices
      if(newLeafs != 0)
        handleLeafs<<<divup(newLeafs,BLOCKSIZE), BLOCKSIZE>>>(
           d_leaves + numleafs, newLeafs);

      numnodes += activenodes;
      numleafs += newLeafs;
      activenodes = newNodes;
      ++cdepth;
    }


    //write triangles
    size_t tris = runPrefixSum(d_triangle_offset, numnodes);
    handleNodes<<<divup(numnodes,BLOCKSIZE), BLOCKSIZE>>>(
       d_activeset, d_triangle_offset, numnodes);
    cudaDeviceSynchronize();
    double dt = timing.end();
    std::cout << "CUDA: " << dt*1000 << " (" << numnodes << "
       nodes)" << std::endl;
    return 3*tris;
  }

};
```

Listing A.4: Softshell Implementation

```

namespace Softshell
{
  __constant__ OctreeEntry* octree;
  __constant__ unsigned int *vertexIdOut, *triOutput;
  __constant__ const unsigned int *vertexCoherence, *
     triangleCoherence;
  __constant__ unsigned int vertices, maxdepth;
  __constant__ const uint3* voxelIds;

  __device__ uint triangleOutCount;
```

```


typedef float Params;
typedef uint2 NodeWorkItem;
typedef CombWorkpackage<NodeWorkItem, Params> NodeWp;


class GenVerticesProcedure: public Procedure
{
public:
  __device__ static void execute(volatile Workpackage*
     workpackage)
  {
    volatile NodeWp* mywp = static_cast<volatile NodeWp*>(
       workpackage);

    uint mynodeid = mywp->getMyWorkItem().x;
    uint mynodedepth = mywp->getMyWorkItem().y;

    const OctreeEntry* node = octree + mynodeid;

    uint vertexTargetStart = node->vertexOffset;
    uint vertexTargetEnd = vertexTargetStart + node->numVertices
       ;
    bool leafnode = mynodedepth >= maxdepth;
    for(uint targetVertex = vertexTargetStart; targetVertex <
       vertexTargetEnd; ++targetVertex)
    {
      uint coherenceId = vertexCoherence[targetVertex];
      vertexIdOut[coherenceId] = leafnode?coherenceId:(vertices
         + mynodeid);
    }

    if(__threadId() == 0)
      delete mywp;
  }
};

template<uint DepthError>
class TraverseProcedure : public Procedure
{
  __device__ static void  writeTriangleOutput(const OctreeEntry*
      node )
  {
    ___shared___ uint myshared[BlockSize+1];
    uint triOffset = 0, numTris = 0;
    if(node != 0)
    {
      triOffset = node->triangleOffset;
      numTris = node->numTriangles;
    }
```

```
      myshared[__threadId()] = numTris;
      //run prefix sum to get local offset
      scan_workefficient(myshared, BlockSize);
      if(__threadId() == BlockSize-1)
        myshared[BlockSize] = atomicAdd(&triangleOutCount,
           myshared[__threadId()]+numTris);
      __sync();

      if(numTris == 0)
        return;

      uint startWrite = myshared[BlockSize] + myshared[__threadId
         ()];

      numTris += triOffset;
      uint * indexoutput = triOutput + 3*startWrite;


      //run through all triangles
      for(uint i = triOffset; i < numTris; ++i)
      {
        //all 3 vertices
        for(unsigned int v = 0; v < 3; ++v)
          *indexoutput++ = vertexIdOut[triangleCoherence[3*i + v
             ]];
      }
    }

  public:

    __device__ static void execute(volatile Workpackage*
       workpackage)
    {
      volatile NodeWp* mywp = static_cast<volatile NodeWp*>(
         workpackage);

      const float threshold = *mywp->params;
      const OctreeEntry* node = 0;

      if(mywp->hasWork())
      {
        uint mynodeid = mywp->getMyWorkItem().x;
        uint mynodedepth = mywp->getMyWorkItem().y;

        uint childDepth = mynodedepth + 1;
        const OctreeEntry* node = octree + mynodeid;

        //check children to be expanded
        uint numChildren = OctreeEntry::hasChildren(node->children
           )?OctreeEntry::getNumChildren(node->children):0;
      uint firstChildId = OctreeEntry::getFirstChild(node->children);
      for(uint i = 0; i < numChildren; ++i)
      {
        uint thisChildId = firstChildId + i;
        const OctreeEntry *childEntry = octree + thisChildId;
        float3 vMin, vMax;
        getVoxelBounds(voxelIds[thisChildId], childDepth, vMin, vMax);
        //check if criterion is fullfilled
        bool expand = carryOnTraversal<DepthError>(childDepth, vMin, vMax, threshold);
        if(expand && childDepth < maxdepth)
          issueWorkItem<TraverseProcedure>(make_uint2(thisChildId, childDepth), mywp->params);
        else //leaf
          issueWorkItem<GenVerticesProcedure>(make_uint2(thisChildId, childDepth), mywp->params);
      }
      if(numChildren == 0) //add vertices for this node
        issueWorkItem<GenVerticesProcedure>(make_uint2(thisChildId, childDepth), mywp->params);
    }

    writeTriangleOutput(node);

    __sync();

    if(__threadId() == 0)
      delete mywp;
  }
  static const uint BlockSize = 256;
};

//Priority class telling the system to execute TraverseProcedures first
class TraversePriority
{
public:
  static __device__ float eval(Workpackage* workpackage, ProcedureInfo* proc)
  {
    //note: the workpackage wp will be called for the procedue proc
    if(procEqual<TraverseProcedure>(*proc))
      return 1.0f;
    else
      return 0.0f;
  }
};

```

```
  template<uint DepthError>
  class TraverseEvent : public EventCreator
  {
  public:
    __device__ static void create(Params* params, void* &storage)
    {
      NodeWorkItem root = make_uint2(0,0);
      emitWorkpackage<TraverseProcedure<DepthError> >(new NodeWp(&
          root, 1, params));
    }
  };

};
```

Listing A.5: Softshell Control

```


#include "softshell.cu"
#include "tools/utils.h"
#include "timing.h"
#include "clustering.h"
#include "helpers.cuh"
#include "scheduled_updateNode.cuh"

namespace Softshell
{
  Timing timing;

  Scheduler scheduling;

  template<uint ERROR_DEPTH>
  size_t buildMesh(CUDAStorages* memory, float threshold, Event<
      TraverseEvent<ERROR_DEPTH> >* useevent);

  Event<TraverseEvent<0> > *buildEvent0;
  Event<TraverseEvent<1> > *buildEvent1;

  unsigned int usedDevice;

  void initScheduling(int device)
  {
    usedDevice = device;
    scheduler.init<TraversePriority>(usedDevice);

    buildEvent0 = scheduler.createEvent<TraverseEvent<0> >();
    buildEvent1 = scheduler.createEvent<TraverseEvent<1> >();
  }
  void shutdownScheduling()
  {
    scheduler.shutdown();
```

```
}

size_t buildMesh(CUDAStorages* memory, float threshold, bool error_depth)
{
  if(error_depth)
    return buildMesh<1>(memory, threshold, buildEvent0);
  else
    return buildMesh<0>(memory, threshold, buildEvent1);
}

void prepareData(CUDAStorages* memory)
{
   setOctreeBoundaries(memory->boundaries);
   setLevelOffsets(memory->h_levelOffset.get());

   CUDA_CHECKED_CALL(cudaMemcpyToSymbol(octree, &memory->d_octree_data, sizeof(OctreeEntry*)));
   CUDA_CHECKED_CALL(cudaMemcpyToSymbol(vertexCoherence, &memory->d_octree_vertex_coherence, sizeof(const unsigned int*)));
   CUDA_CHECKED_CALL(cudaMemcpyToSymbol(vertexIdOut, &memory->d_vertex_lookup, sizeof(unsigned int *)));
   CUDA_CHECKED_CALL(cudaMemcpyToSymbol(vertices, &memory->vertexcount, sizeof(unsigned int)));
   CUDA_CHECKED_CALL(cudaMemcpyToSymbol(maxdepth, &memory->maxdepth, sizeof(unsigned int)));
   CUDA_CHECKED_CALL(cudaMemcpyToSymbol(voxelIds, &memory->d_voxel_ids, sizeof(const uint3*)));
   CUDA_CHECKED_CALL(cudaMemcpyToSymbol(triangleCoherence, &memory->d_octree_triangle_coherence, sizeof(const unsigned int*)));
}

template<uint ERROR_DEPTH>
size_t buildMesh(CUDAStorages* memory, float threshold, Event<TraverseEvent<ERROR_DEPTH> >* useevent)
{
  timing.begin();

  uint null = 0;
  CUDA_CHECKED_CALL(cudaMemcpyToSymbol(triangleOutCount, &null, sizeof(uint)));
  CUDA::GL::mapped_buffer<unsigned int> d_indices(memory->cuda_index_buffer);
  CUDA_CHECKED_CALL(cudaMemcpyToSymbol(triOutput, &d_indices.get(), sizeof(uint*)));

  useevent->fire(threshold);
  useevent->join();

```

```
    uint triangles;
    CUDA_CHECKED_CALL(cudaMemcpyFromSymbol(&triangles,
        triangleOutCount, sizeof(uint)));

    double dt = timing.end();
    std::cout << "Softshell: " << dt*1000 << " (" << triangles <<
        " triangles)" << std::endl;

    return 3*triangles;
  }
};
```

A.2 Path Tracing

This sections shows the source code of the comparison implementation between Softshell and CUDA for the path tracing test case.

Listing A.6: Core Functions

```
#ifndef COREFUNCTIONS_CUH
#define COREFUNCTIONS_CUH

#include "structures.h"

__device__ float device_random( uint& seed )
{
  const uint MULTIPLIER = 1664525;
  const uint OFFSET = 1013904223;
  const double MODULUS = 4294967295.0;
  const float MODULUS_INV = (float) (1.0 / MODULUS);

  seed = seed * MULTIPLIER + OFFSET;
  float res = seed * MODULUS_INV;
  return res;
}

__device__ bool intersectSphere(const object& sphere, float3 &d,
    float3 &n, float& mindist, const float3& orig, const float3&
    dir)
{
  d = sphere.pos - orig;
  float v = vec_dot( dir, d );

  if( v - sphere.rad > mindist )
    return false;

  float t = sphere.rad*sphere.rad + v*v - d.x*d.x - d.y*d.y - d.z*
      d.z;
  if( t < 0 )
    return false;

  t = v - sqrt( t );
  if( ( t > mindist ) || ( t < 0 ) )
    return false;

  n = orig + t*dir - sphere.pos;
  n = vec_normalize( n );

  mindist = t;
  return true;
}

__device__ bool intersectPlane(const object& plane, vector3 &d,
    vector3 &n, float& mindist, const vector3& orig, const vector3&
     dir)
{
  float v = vec_dot( plane.n, dir );

  if( v >= 0 )
    return false;

  d = plane.pos - orig;

  float t = vec_dot( plane.n, d ) / v;
  if( ( t > mindist ) || ( t < 0 ) )
    return false;

  n = plane.n;
  mindist = t;
  return true;
}

__device__ void hitObject(const object& hitObject, float t, const
    vector3 &n, vector3& orig, vector3& dir, vector3& contrib,
    rgbcolor& color, uint& seed)
{
  const float EPSILON = 0.001f;
  vector3 hit = orig + t*dir;

  t = 2.0f * vec_dot( -dir, n );
  dir = t*n+dir;
  dir = vec_normalize( dir );

  vector3 o;
  do
  {
    o = 2.0f*make_float3(device_random(seed),device_random(seed),
        device_random(seed))-1.0f;
  }
  while( ( vec_dot(o,o) > 1 ) || ( vec_dot( o, n ) <= 0 ) );

```

```
    float v = (hitObject.d*device_random(seed))*hitObject.g +
        hitObject.d*(1.0f-hitObject.g);
    dir = o*v + dir*(1.0f-v);
    dir = vec_normalize( dir );

    orig = hit + EPSILON*dir;

    contrib *= hitObject.c;

    color += hitObject.e * contrib;
}

__device__ const object* intersectObjects(const object* objects,
    uint numobjects, float3 &d, float3 &n, float& t, const float3&
    orig, const float3& dir)
{
    t = 1000000.0f;
    const object* res = 0;
    for(uint i = 0; i < numobjects; ++i)
    {
        if(objects[i].type == TYPE_SPHERE)
        {
            if(intersectSphere(objects[i], d, n, t, orig, dir))
                res = objects + i;
        }
        else if(objects[i].type == TYPE_PLANE)
        {
            if(intersectPlane(objects[i], d, n, t, orig, dir))
                res = objects + i;
        }
    }
    return res;
}

#endif
```

Listing A.7: Cuda Tracer

```
#include <cuda.h>
#include <cutil.h>
#include <cutil_math.h>

#include "scene.h"
#include "structures.h"

#include <cuda.h>
#include "renderingdata.cuh"
#include "corefunctions.cuh"

#include "timing.h"

```

```


__global__ void startTrace(rgbcolor* imagedata, uint* randseeds);


double traceCuda(int device, const Scene& scene)
{
  cudaSetDevice(device);
  setupData(scene);
  initImage();
  initRand();

  double t0 = Timing::gettime();
  startTrace<<<scene.imgDims(), tsamples>>>(d_imagedata,
      d_randseeds);
  cudaDeviceSynchronize();
  double t1 = Timing::gettime();
  return t1 - t0;
}




__global__ void startTrace(rgbcolor* imagedata, uint* randseeds)
{
  uint2 pixel = make_uint2( blockIdx.x, blockIdx.y);
  unsigned int randseed =  getSeed(randseeds, pixel, threadIdx.x);

  vector3 contrib = make_float3(1.0f);
  vector3 orig = make_float3(0.0f);
  rgbcolor color = make_float3(0.0f);

  vector3 dir;
  dir.x = ( (float)pixel.x / (float)c_width ) - 0.5f +
      device_random(randseed)/(float)c_width;
  dir.y = ( ( (float)pixel.y / (float)c_height ) - 0.5f ) * ( (
      float)c_height/(float)c_width) + device_random(randseed)/(
      float)c_height;
  dir.z = 1;
  dir = vec_normalize( dir );


  uint recursion = 0;
  while( ( recursion < c_maxrecursions ) && ( ( contrib.x >
      MIN_CONTRIB ) || ( contrib.y  > MIN_CONTRIB ) || ( contrib.z
      > MIN_CONTRIB ) ) )
  {
    float3 d, n;
    float t;
    const object* obj = intersectObjects(c_objects, c_numObjects,
        d, n, t, orig, dir);

    if(obj)
      hitObject(*obj, t, n, orig, dir, contrib, color, randseed);
    else
    {
      color += c_background * contrib;
      contrib.x = contrib.y = contrib.z = 0;
    }
    recursion++;
  }

  rgbcolor* outcolor = imagedata + (pixel.y*c_width + pixel);
  atomicAdd(&outcolor->x, color.x);
  atomicAdd(&outcolor->y, color.y);
  atomicAdd(&outcolor->z, color.z);
}
```

Listing A.8: Softshell Tracer

```

#include "gpuscheduling.cu"

#include "image.h"
#include "renderingdata.cuh"
#include "corefunctions.cuh"

#include "timing.h"

struct Params
{
    unsigned int* p_randseeds;
    rgbcolor* p_imagedata;
}

struct TraceWp : public PayloadedWorkpackage<Params>
{
  uint2 pix;
  TraceWp(uint3 _pix, uint threads, Params* _params) :
    pix(_pix.x, _pix.y),
    PayloadedWorkpackage<Params>(threads, _params)
  { }
};


class TracingProcedure : public Procedure
{
public:
  //macro will activate the third tier for this procedure
  ACTIVATE_THIRD_TIER

```

```
__device__ static void execute(volatile Workpackage* workpackage
  )
{

  //define additional payload directly in execute
  PAYLOAD_BEGIN
  vector3 contrib;
  vector3 orig;
  rgbcolor color;
  vector3 dir;
  unsigned int randseed;
  uint recursion;
  PAYLOAD_END

  volatile TraceWp* mywp = static_cast<volatile TraceWp*>(
      workpackage);
  uint2 pixel = mywp->pix;

  //init data structures
  randseed =  getSeed(mywp->params->p_randseeds, pixel,
      __threadId());

  contrib = make_float3(1.0f);
  orig = make_float3(0.0f);
  color = make_float3(0.0f);

  dir.x = ( (float)pixel.x / (float)c_width ) - 0.5f +
      device_random(randseed)/(float)c_width;
  dir.y = ( ( (float)pixel.y / (float)c_height ) - 0.5f ) * ( (
      float)c_height/(float)c_width) + device_random(randseed)/(
      float)c_height;
  dir.z = 1;
  dir = vec_normalize( dir );

  recursion = 0;

  //scheduler is activated every 10th iteration of this loop
  SCHEDULEDWHILE(true, 10)
  {
    if( !( ( recursion < c_maxrecursions ) && ( ( contrib.x >
        MIN_CONTRIB ) || ( contrib.y  > MIN_CONTRIB ) || (
        contrib.z > MIN_CONTRIB ) ) ) )
    {
      rgbcolor* outcolor = mywp->params->p_imagedata + (pixel.y*
          c_width + pixel.x);
      atomicAdd(&outcolor->x, color.x);
      atomicAdd(&outcolor->y, color.y);
      atomicAdd(&outcolor->z, color.z);
      SCHEDULEENDTHREAD;
    }

```

```

      float3 d, n;
      float t;
      const object* obj = intersectObjects(c_objects, c_numObjects, d, n, t, orig, dir);

      if(obj)
        hitObject(*obj, t, n, orig, dir, contrib, color, randseed);
      else
      {
        color += c_background * contrib;
        contrib.x = contrib.y = contrib.z = 0;
      }
      recursion++;
    }
    SCHEDULEDWHILEEND
  }
};

double renderScheduled(int device, const Scene& scene)
{
  Scheduler scheduler(device);

  setupData(scene);
  initImage();
  initRand();

  Params params;
  params.p_imagedata = d_imagedata;
  params.p_randseeds = d_randseeds;

  auto tracingEvent = scheduler.createEvent<TraceWp, TracingProcedure, Params>();

  double t0 = Timing::gettime();

  dim imageDims(scene.imgWidth(), scene.imgHeight());
  tracingEvent->fire(DefaultCreatorParams(imageDims, tsamples, params));
  TracingEvent->join();

  double t1 = Timing::gettime();
  return t1-t0;
}


#endif
```

Appendix B

Massively Parallel Queuing Source Code

Listing B.1: Mutex Queue

```
class BlockingQueue
{
  Mutex mutex;
  volatile uint front;
  volatile uint back;

  volatile Data buffer[QueueSize];
public:
  __device__ bool enqueue(Data data) {
    while(true) {
      int mypos = __popc(lanemask_lt() & __ballot(1));
      if(mypos == 0) {
        mutex.acquire();

        uint pos = back;
        if(pos + 1 - front > QueueSize) {
          mutex.release();
          return false;
        }

        buffer[pos%QueueSize] = data;
        __threadfence();

        mutex.release();
        return true;
      }
    }
  }

  __device__ int dequeue(Data* data, int num)
  {
    __shared__ int offset, take;

    if(threadIdx.x == 0) {
      mutex.acquire();
      offset = front;
      take = min(num, back - offset);
      if(take > 0)
        front = pos + canTake;
    }
    __syncthreads();
    if(threadIdx.x < take) {
      data[threadIdx.x] = buffer[(offset + threadIdx.x)%QueueSize];
      __threadfence();
    }
    __syncthreads();
    if(threadIdx.x == 0)
      mutex.release();
    return take;
  }
};
```

Listing B.2: Dual Mutex Queue

```
class BlockingQueue
{
  Mutex front_mutex;
  Mutex back_mutex;
  volatile uint front;
  volatile uint back;

  volatile Data buffer[QueueSize];
public:
  __device__ bool enqueue(Data data) {
    while(true) {
      int mypos = __popc(lanemask_lt() & __ballot(1));
      if(mypos == 0) {
        back_mutex.acquire();

        uint pos = back;
        if(pos + 1 - front > QueueSize) {
          mutex.release();
          return false;
        }

        buffer[pos%QueueSize] = data;
        __threadfence();
        ++back;
        __threadfence();

        back_mutex.release();
        return true;
      }
    }
  }

  __device__ int dequeue(Data* data, int num)
  {
    __shared__ int offset, take;

    if(threadIdx.x == 0) {
      front_mutex.acquire();
      offset = front;
      take = min(num, back - offset);
    }
    __syncthreads();
    if(threadIdx.x < take) {
      data[threadIdx.x] = buffer[(offset + threadIdx.x)%QueueSize
          ];
      __threadfence();
    }
    __syncthreads();
    if(threadIdx.x == 0) {
      front = pos + canTake;
      __threadfence();
      mutex.release();
    }
    return take;
  }
};
```

Listing B.3: CAS Ordered Queue

```
class CASOrderingQueue
{
 volatile uint backWrite;
 volatile uint back;
 volatile uint frontRead;
 volatile uint front;


 volatile Data buffer[QueueSize];
public:

 __device__ bool enqueue(Data data) {
  uint pos = atomicAdd(&backWrite, 1);
  while(pos - front > QueueSize)
   if(atomicCAS(&backWrite, pos + 1, pos) == pos + 1)
    return false;

  buffer[pos%QueueSize] = data;
  __threadfence();

  while(atomicCAS(&back, pos, pos + 1) != pos)
   __threadfence();

  return true;
 }

 __device__ int dequeue(Data* data, int num)
 {
  __shared__ int offset, take;

  if(threadIdx.x == 0) {
   uint f = frontRead;
   uint b = back;
   take = 0;
   while(b > f) {
    uint canTake = min(num, b - f);
    uint pos = atomicCAS((uint*)&frontRead, f, f + canTake);
    if(pos == f) {
     take = canTake;
     offset = pos;
     break;
    } else {
     f = pos;
     b = back;
     __threadfence();
    }
   }
  }
  __syncthreads();
  if(threadIdx.x < take) {
   data[threadIdx.x] = buffer[(offset + threadIdx.x)%QueueSize];
   __threadfence();
  }
  __syncthreads();
  if(threadIdx.x == 0 && take != 0)
   while(atomicCAS(&front, offset, offset + take) != offset)
    __threadfence();
  return take;
 }
};
```

Listing B.4: Distributed Locks Queue

```
class DistLockedQueue
{
 uint front, back;
 volatile int count;
 volatile uint locks[QueueSize];
 volatile Data buffer[QueueSize];
public:
 __device__ bool enqueue(Data data) {
  int c = atomicAdd(&count, 1);
  if(c + 1 < QueueSize) {
   uint pos = atomicAdd(&back, 1) % QueueSize;
   while(locks[pos] != 0)
```

```
      __threadfence();
    buffer[pos] = data;
    __threadfence();
    locks[pos] = 1;
    return true;
   }
   atomicSub(&count, 1);
   return false;
  }

  __device__ int dequeue(Data* data, int num)
  {
   __shared__ int offset, take;

   if(threadIdx.x == 0) {
    int c = atomicSub(&count, num);
    if(c < num) {
     atomicAdd(&count, min(num,num - c));
     num = max(c, 0);
    }
    take = num;
    if(num > 0)
     offset = atomicAdd(&front, num);
   }
   __syncthreads();
   if(threadIdx.x < take)
   {
    uint p = (offset + threadIdx.x)%QueueSize;
    while(locks[p] != 1)
     __threadfence();
    data[threadIdx.x] = buffer[p];
    __threadfence();
    locks[p] = 0;
   }
   return take;
  }
};
```

Listing B.5: Pointed Queue

```
class PointedQueue
{
 uint front, back;
 volatile int count;
 Data* volatile datapointers[QueueSize];
public:
 __device__ bool enqueue(Data data) {
  int c = atomicAdd(&count, 1);
  if(c + 1 < QueueSize) {
   Data* globalData = allocate(data);
   __threadfence();
   uint pos = atomicAdd(&back, 1) % QueueSize;
   while(datapointers[pos] != nullptr)
    __threadfence();
   datapointers[pos] = data;
   return true;
  }
  atomicSub(&count, 1);
  return false;
 }

 __device__ int dequeue(Data* data, int num)
 {
  __shared__ int offset, take;

  if(threadIdx.x == 0) {
   int c = atomicSub(&count, num);
   if(c < num) {
    atomicAdd(&count, min(num,num - c));
    num = max(c, 0);
   }
   take = num;
   if(num > 0)
    offset = atomicAdd(&front, num);
  }
  __syncthreads();
  if(threadIdx.x < take)
  {
   uint p = (offset + threadIdx.x) % QueueSize;
   while(locks[p] == nullptr)
    __threadfence();
   data[threadIdx.x] = *datapointers[p];
   __threadfence();
   datapointers[p] = nullptr;
  }
  return take;
 }
};
```

Listing B.6: Local Queue

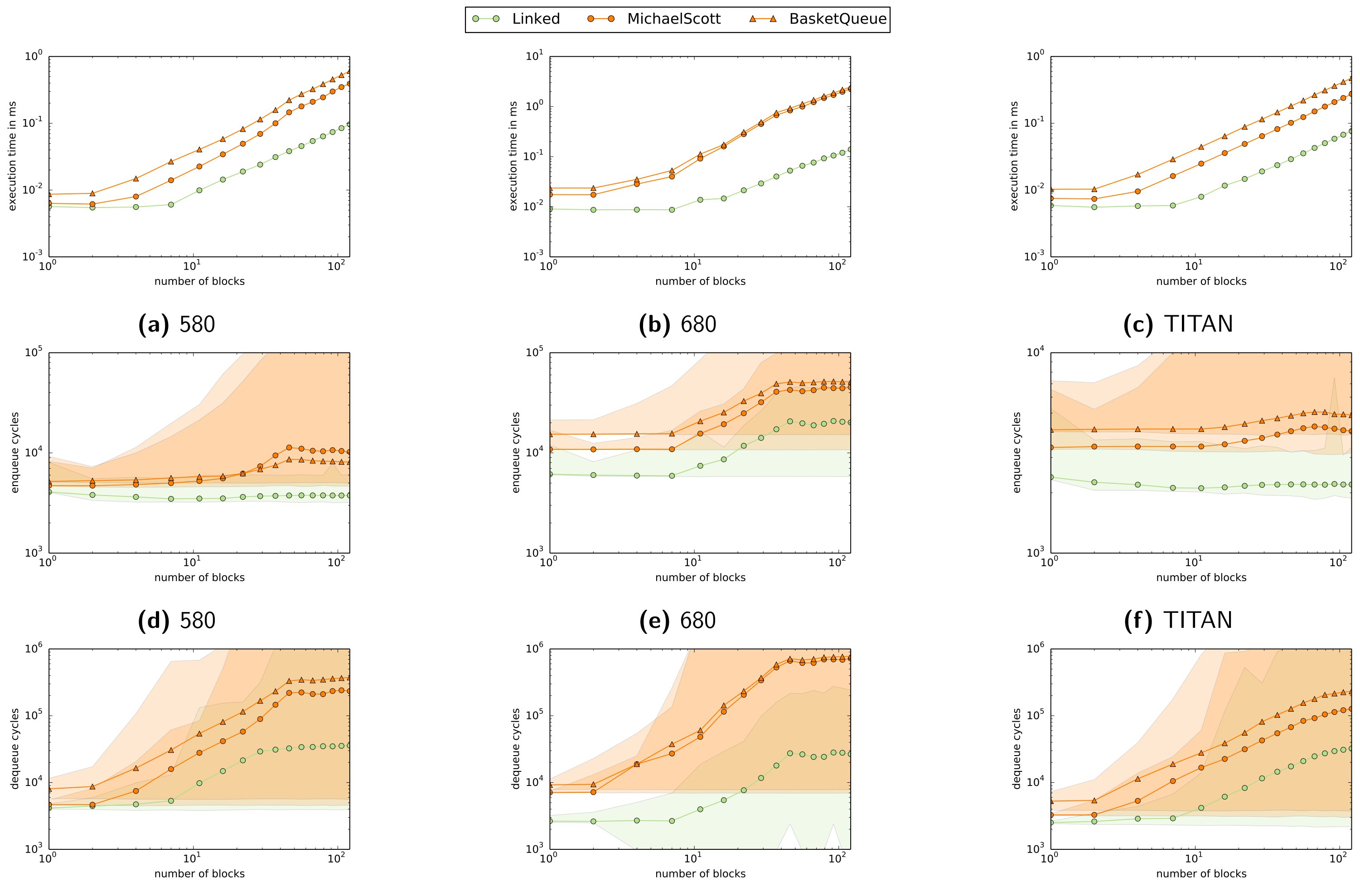
```
class LocalQueue
{
 volatile int counter;
 volatile Data buffer[QueueSize];
public:
 __device__ bool enqueue(Data data) {
   if(counter >= NumElements)
    return false;
   int pos = atomicAdd(&counter, 1);
   if(pos >= NumElements)
     atomicSub(&counter, 1);

    buffer[pos] = data;
    return true;
  }

  __device__ int dequeue(Data* data, int num) {
   int n = counter;
   __syncthreads();
   if(threadIdx.x == 0)
    counter = max(0, n - maxnum);
   int take = min(maxnum, n);
   int offset = n - take;

   if(threadIdx.x < take)
    data[threadIdx.x] = buffer[offset + threadIdx.x];

   return take;
  }
};
```

Listing B.7: Dynamic Local Queue

```
class DynamicLocalQueue
{
 volatile char buffer[MemSize/sizeof(uint)];

 struct Header
 {
  volatile uint num_id;
  volatile int counter;
 }
public:
 __device__ bool enqueue(Data data, int ProcedureId) {
  int header_pos = 0;
  while(header_pos < MemSize) {
   Header* h = reinterpret_cast<Header*>(buffer + hpos);
   int maxhold = 0;

   uint current_procedure = h->num_id;
   if( (current_procedure & 0xFFFF) == ProcedureId)
    maxhold = (current_procedure >> 16)*16/sizeof(Data);
   else if(current_procedure == 0) {
    uint fitsize = (MemSize - hpos - sizeof(Header))/sizeof(Data);
    uint newsize = min(MaxSharedElements, fittingsize);
    uint num_id = ((newsize*sizeof(Data)/16)<<16) | ProcedureId;

    current_procedure = atomicCAS(&h->num_id, 0, num_id);
    if(current_procedure == 0) {
     maxhold = newsize;
     current_procedure = num_id;
    }
   else if(current_procedure == procId_maxnum)
    maxhold = newsize;
  }
  if(h->counter < maxhold) {
   int pos = atomicAdd(&h->counter, 1);
   if(pos >= maxhold)
    atomicSub(&h->counter, 1);
   else {
    reinterpret_cast<Data*>(h + 1)[pos] = data;
    return true;
   }
  }
  int partSize = (current_procedure >> 16)*16 + sizeof(Header);
  hpos += partSize;
 }
}

__device__ int dequeue(Data* data, int num) {
 __shared__ int take, offset;
 if(threadIdx.x == 0)
 {
  take = 0;
  int hpos = 0;
  while(hpos < MemSize)
  {
   Header* h = reinterpret_cast<Header*>(buffer + hpos);
   if(h->num_id == 0)
    break;
   int c = h->counter;
   if(c > 0)
   {
    take = min(c,num);
    int offset = hpos + sizeof(Header) + (c - take)*sizeof(Data);
    h->counter = c - take;
    break;
   }
   int partSize = (current_procedure >> 16)*16 + sizeof(Header);
   hpos += partSize;
  }
 }
 __syncthreads();
 if(threadIdx.x < take)
 {
  Data* outelement = buffer + offset + threadIdx.x * sizeof(Data);
  data[threadIdx.x] = *outelement;
 }
 return take;
}

__device__ int maintain() {
  int hpos = 0;
  int clearpos = 0;
  while(hpos < MemSize)
  {
   Header* h = reinterpret_cast<Header*>(buffer + hpos);
   if(h->maxnum__procId == 0)
    break;

   int thissize = (h->maxnum__procId >> 16)*16 + sizeof(Header);
   if(h->counter != 0)
   {
    //queue section needs to be copied
    if(clearpos != hpos)
    {
     for(int i = 0; i < thissize/sizeof(int); i+=blockDim.x)
     {
      int c;
      if(i + threadIdx.x < thissize/sizeof(int))
       c = *(reinterpret_cast<int*>(buffer) +
         hpos/sizeof(int) + i + threadIdx.x);
      __syncthreads();
      if(i + threadIdx.x < thissize/sizeof(int))
       *(reinterpret_cast<int*>(buffer) +
         clpos/sizeof(int) + i + threadIdx.x) = c;
     }
    }
    clearpos += thissize;
   }
   hpos += thissize;
  }
  __syncthreads();

  //zero the rest
  for(int i = clpos/sizeof(int) + threadIdx.x;
   i < hpos/sizeof(int); i+=blockDim.x)
    *(reinterpret_cast<int*>(buffer) + i) = 0;
  __syncthreads();
 }
};
```

Appendix C

Massively Parallel Queuing Results

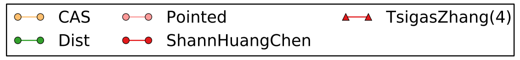


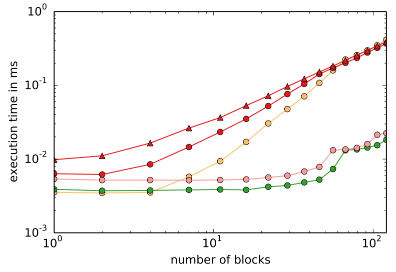
(a) 580 **(b)** 680 **(c)** TITAN

(d) 580 **(e)** 680 **(f)** TITAN

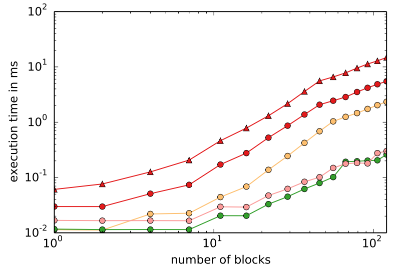
(g) 580 **(h)** 680 **(i)** TITAN

Figure C.1: Linked Queues Package Scenario

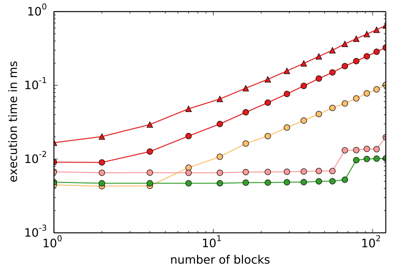




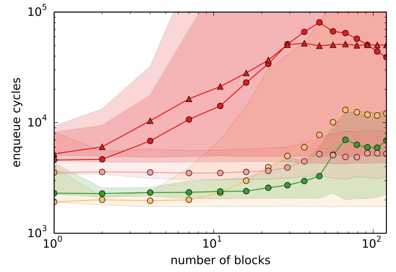
(a) 580



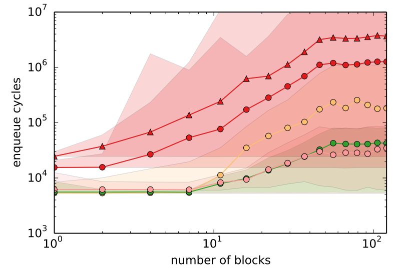
(b) 680



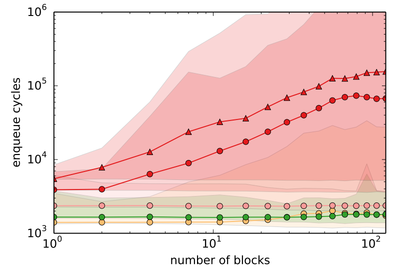
(c) TITAN



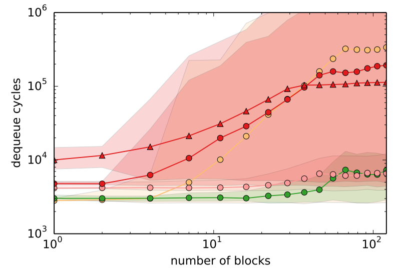
(d) 580



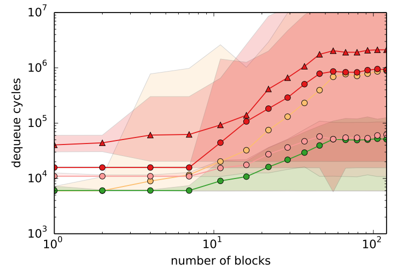
(e) 680



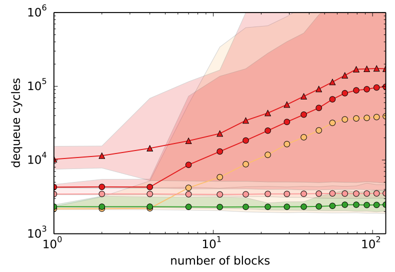
(f) TITAN



(g) 580

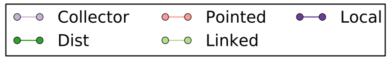


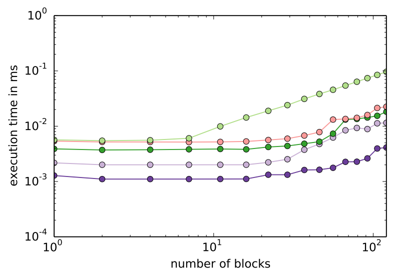
(h) 680



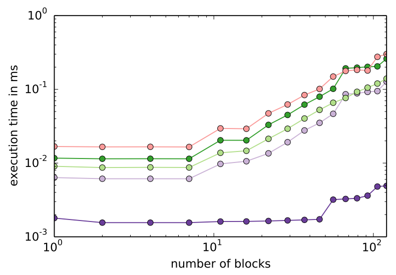
(i) TITAN

Figure C.2: Array Queues Package Scenario

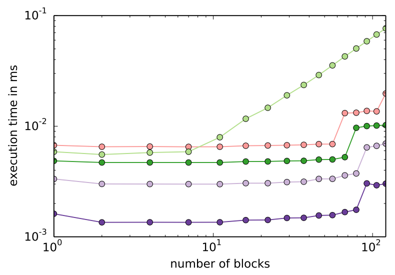




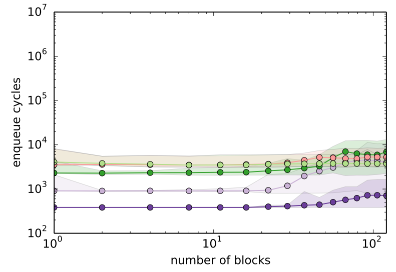
(a) 580



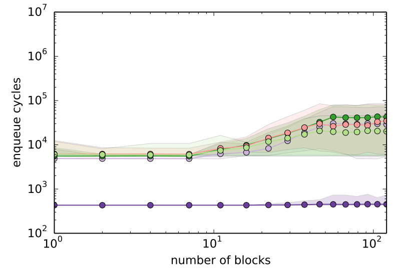
(b) 680



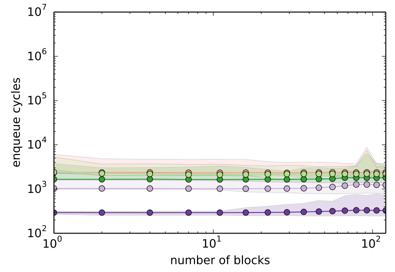
(c) TITAN



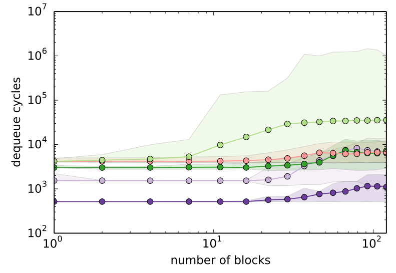
(d) 580



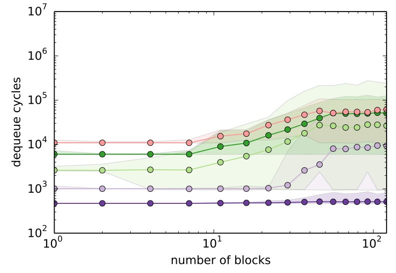
(e) 680



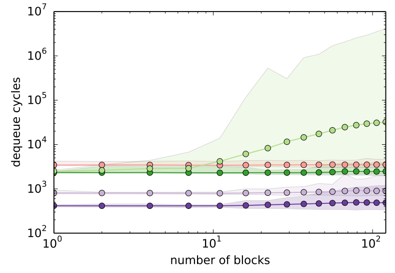
(f) TITAN



(g) 580

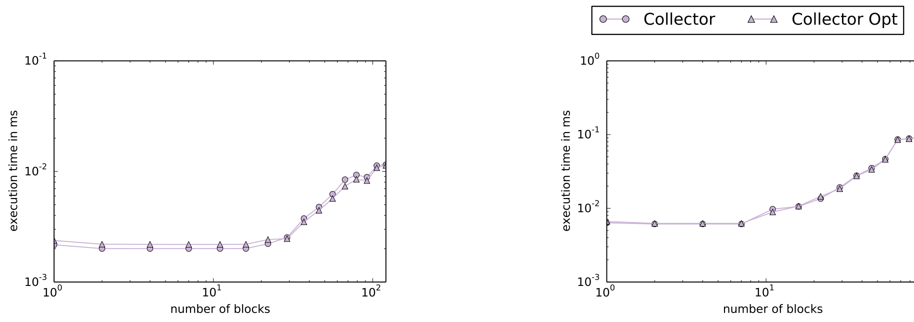


(h) 680

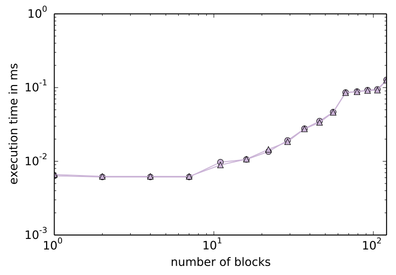


(i) TITAN

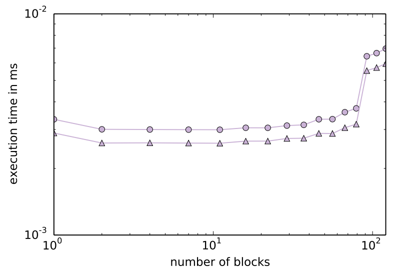
Figure C.3: Overview Package Scenario



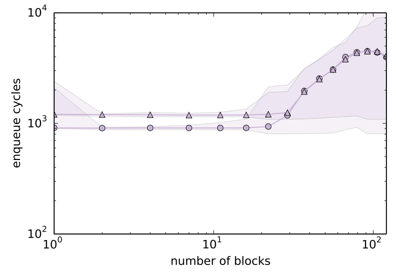
(a) 580



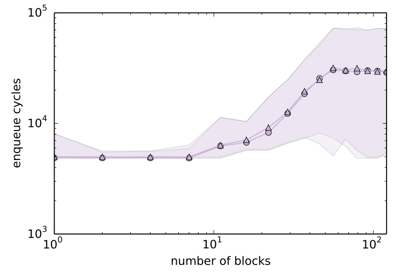
(b) 680



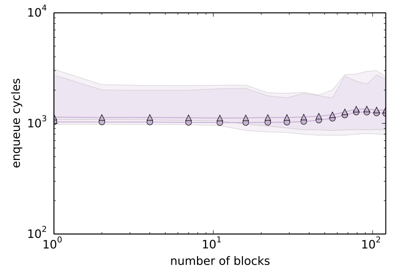
(c) TITAN



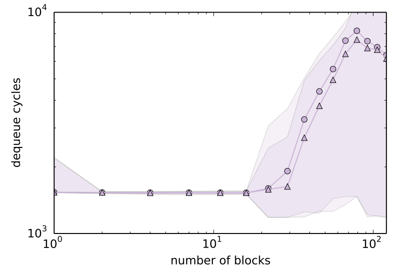
(d) 580



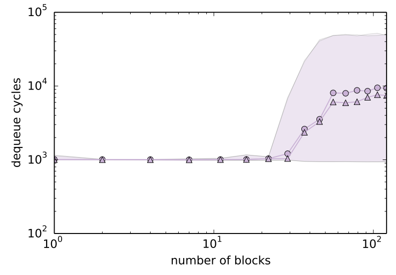
(e) 680



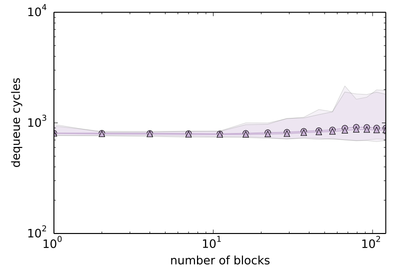
(f) TITAN



(g) 580

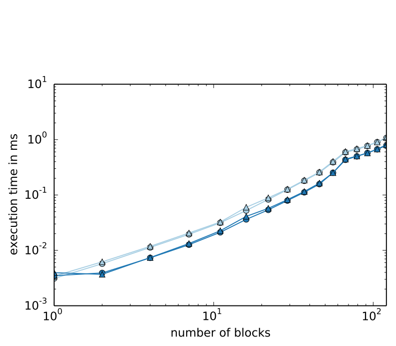


(h) 680

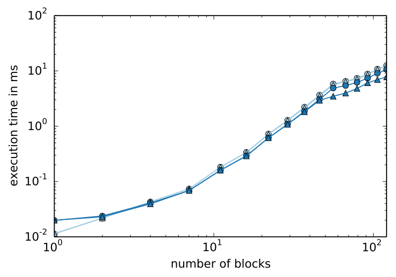


(i) TITAN

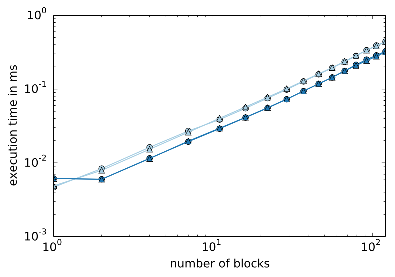
Figure C.4: Collector Queue Package Scenario



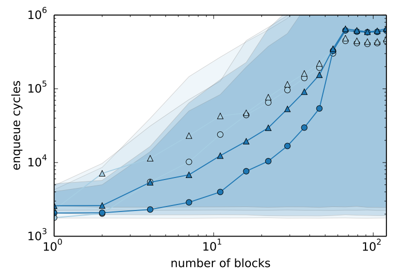
(a) 580



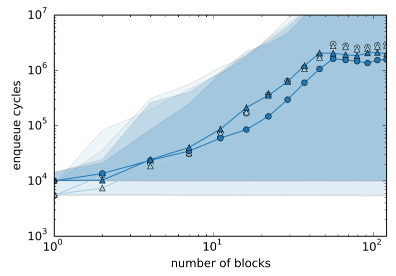
(b) 680



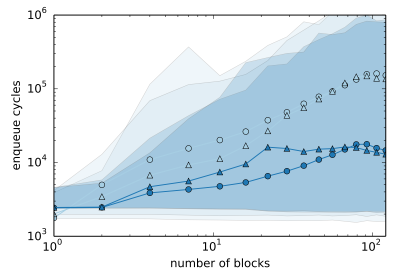
(c) TITAN



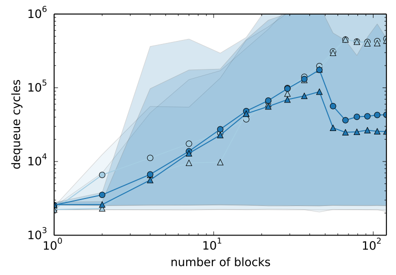
(d) 580



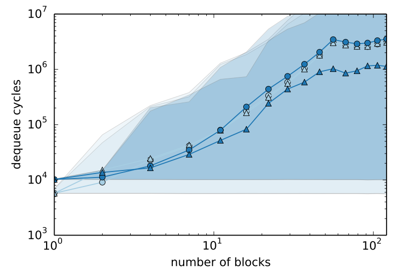
(e) 680



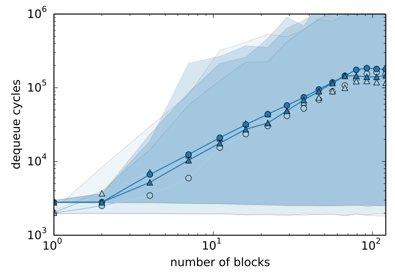
(f) TITAN



(g) 580

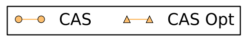


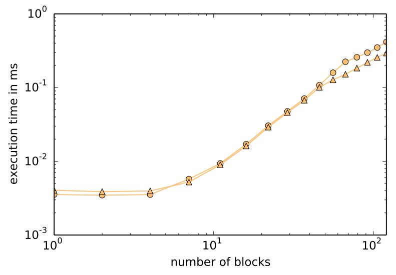
(h) 680



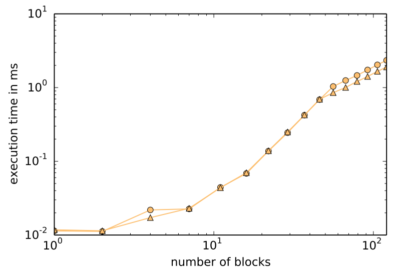
(i) TITAN

Figure C.5: Mutex Queue Package Scenario

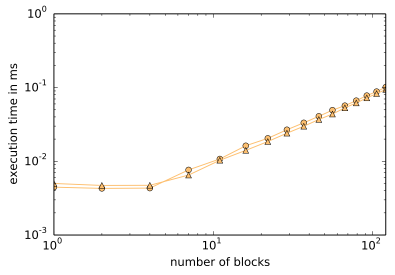




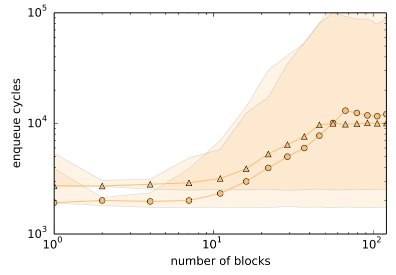
(a) 580



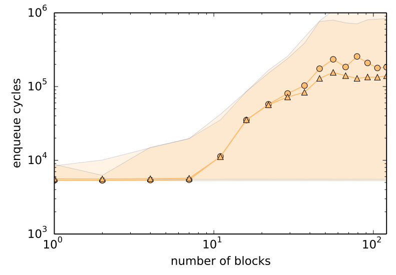
(b) 680



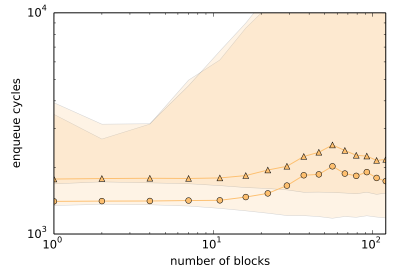
(c) TITAN



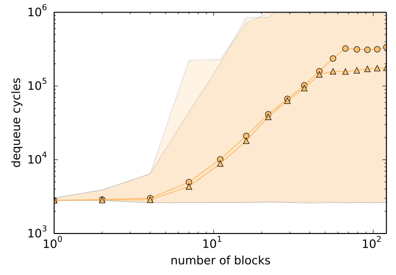
(d) 580



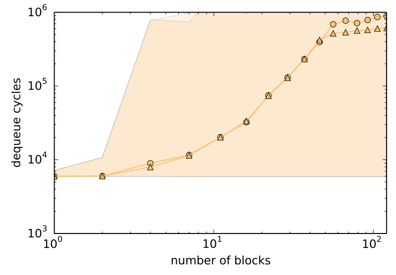
(e) 680



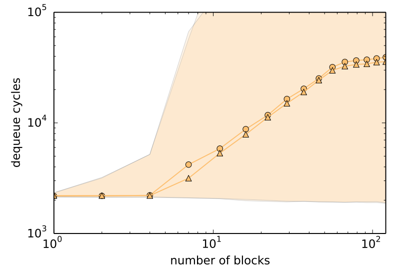
(f) TITAN



(g) 580

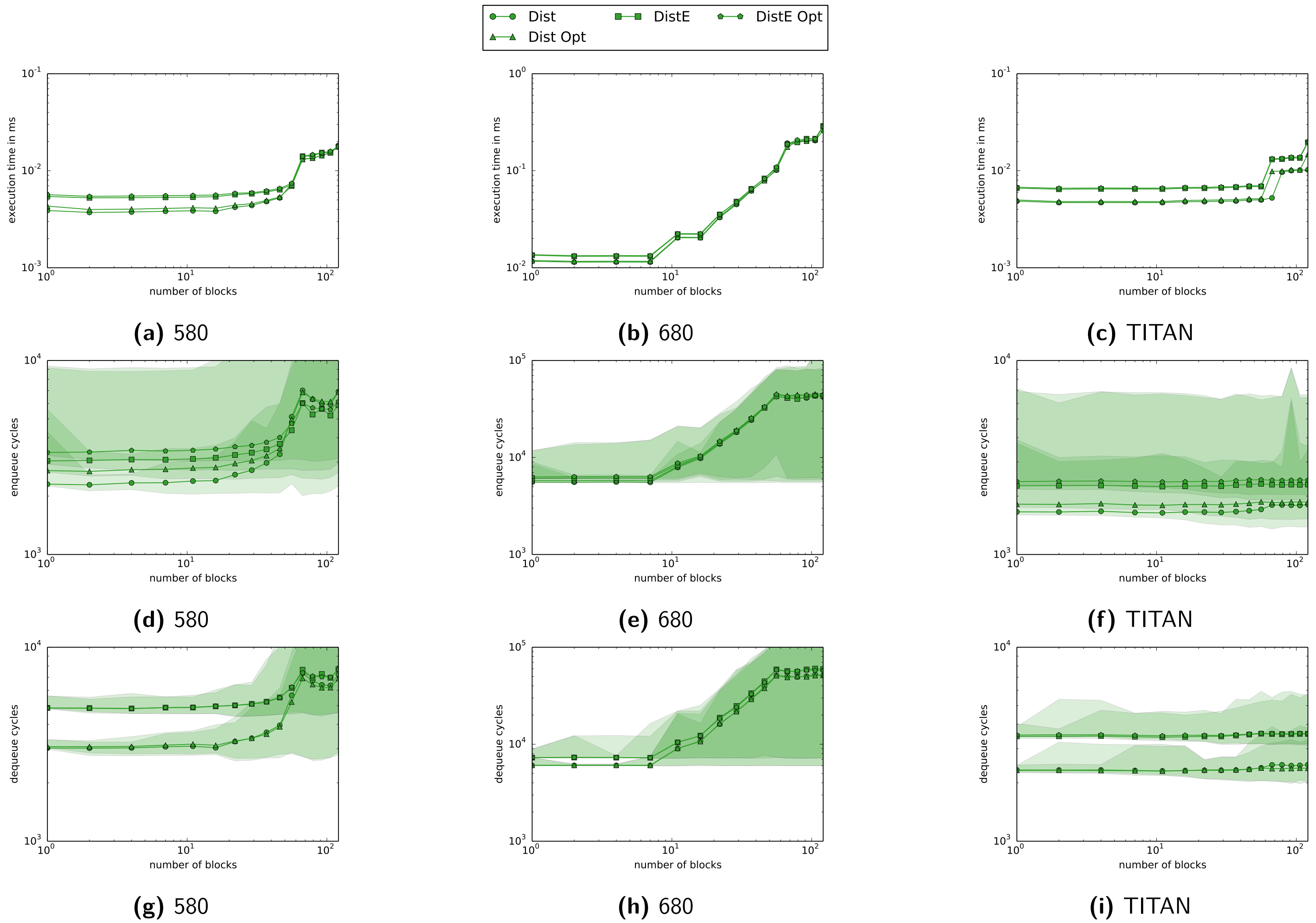


(h) 680



(i) TITAN

Figure C.6: CAS Ordered Queue Package Scenario



(a) 580 (b) 680 (c) TITAN

(d) 580 (e) 680 (f) TITAN

(g) 580 (h) 680 (i) TITAN

Figure C.7: Distributed Locks Queue Package Scenario

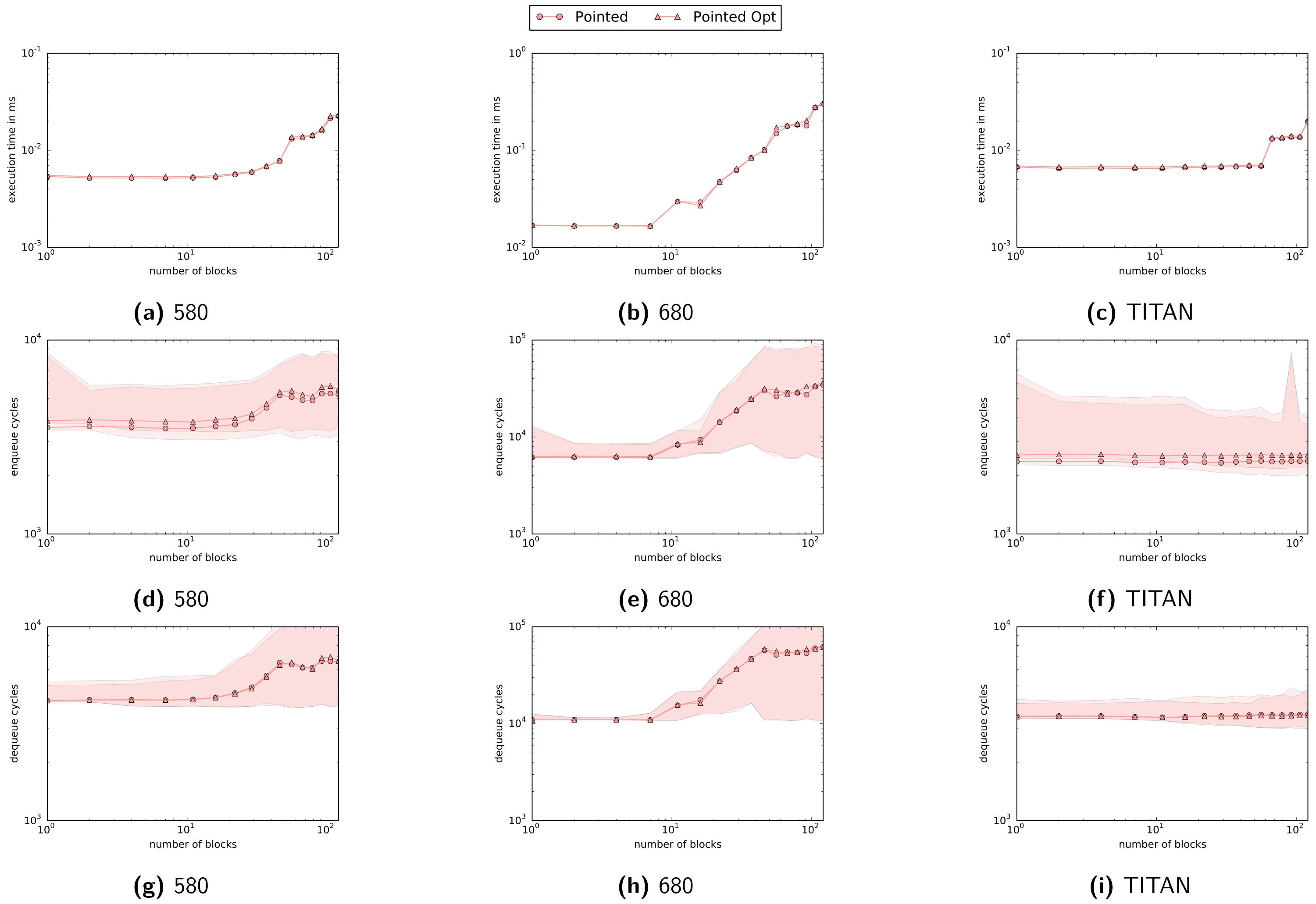
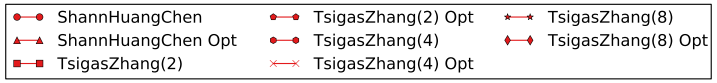
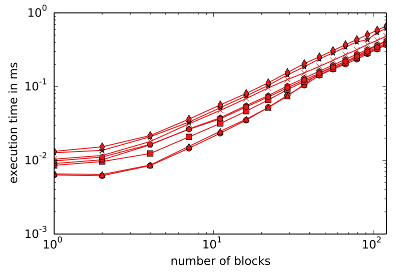
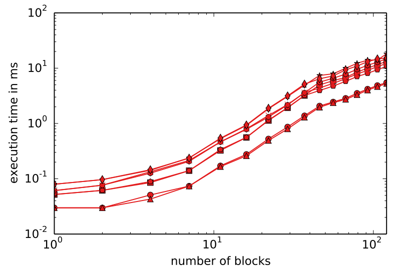


Figure C.8: Pointed Queue Package Scenario

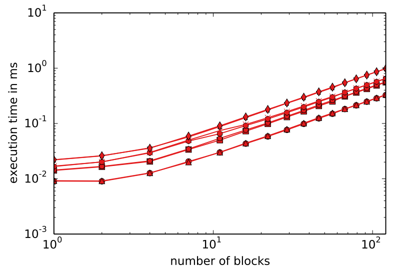




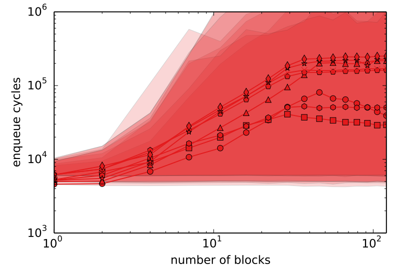
(a) 580



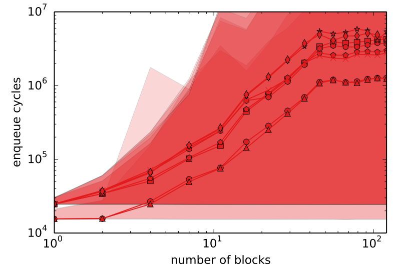
(b) 680



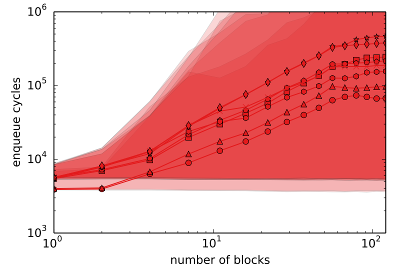
(c) TITAN



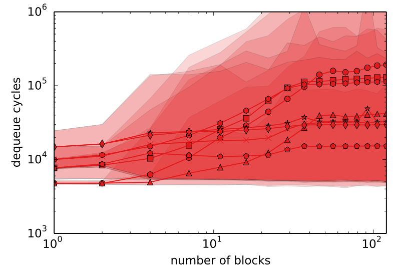
(d) 580



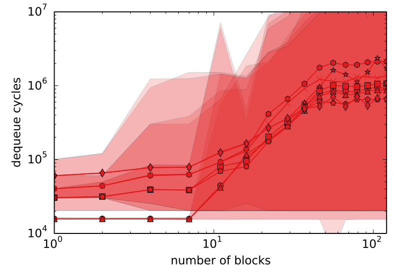
(e) 680



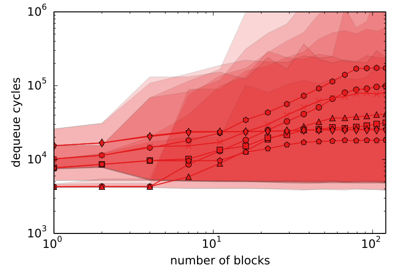
(f) TITAN



(g) 580

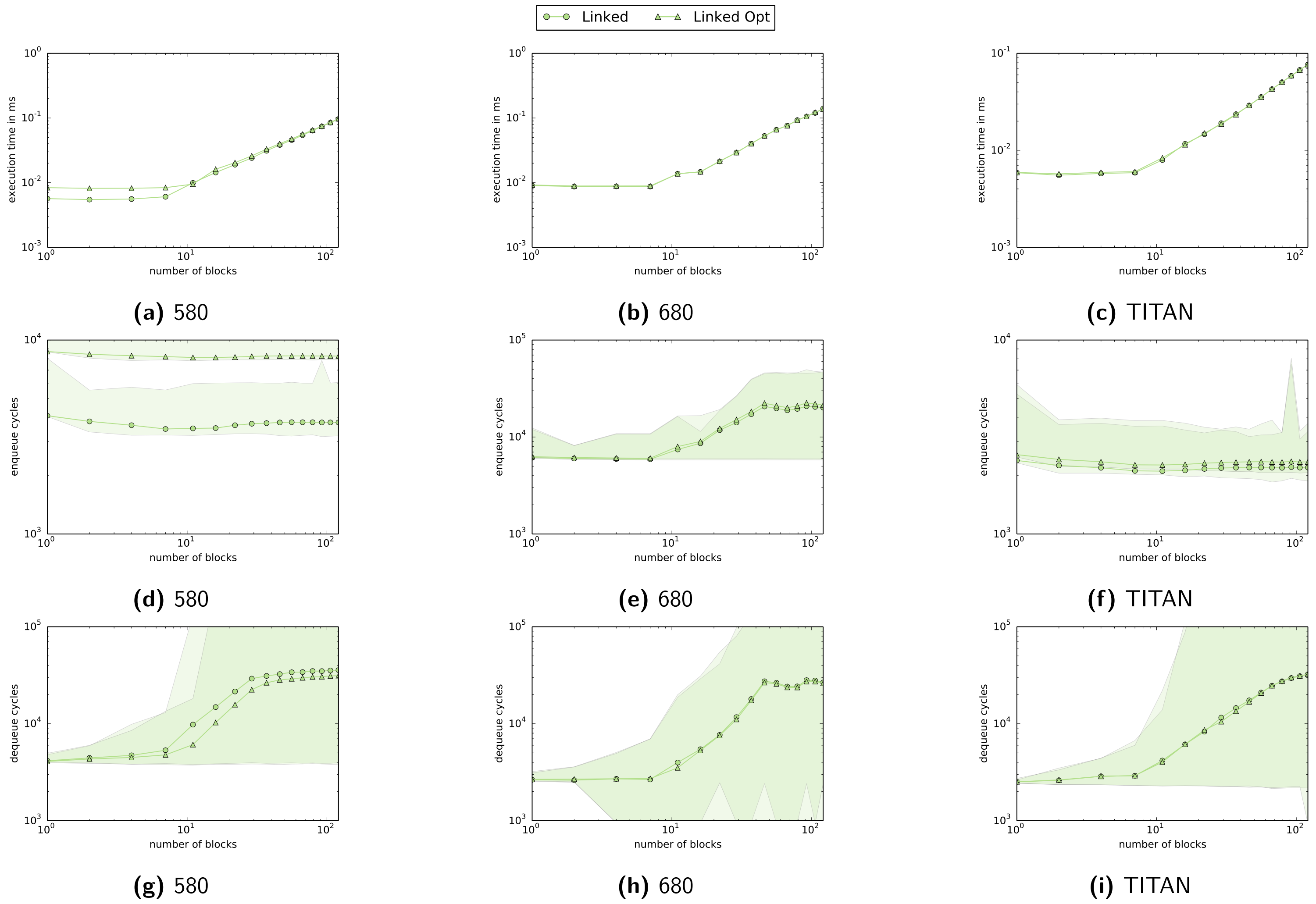


(h) 680



(i) TITAN

Figure C.9: Shan Huang Chen Package Scenario

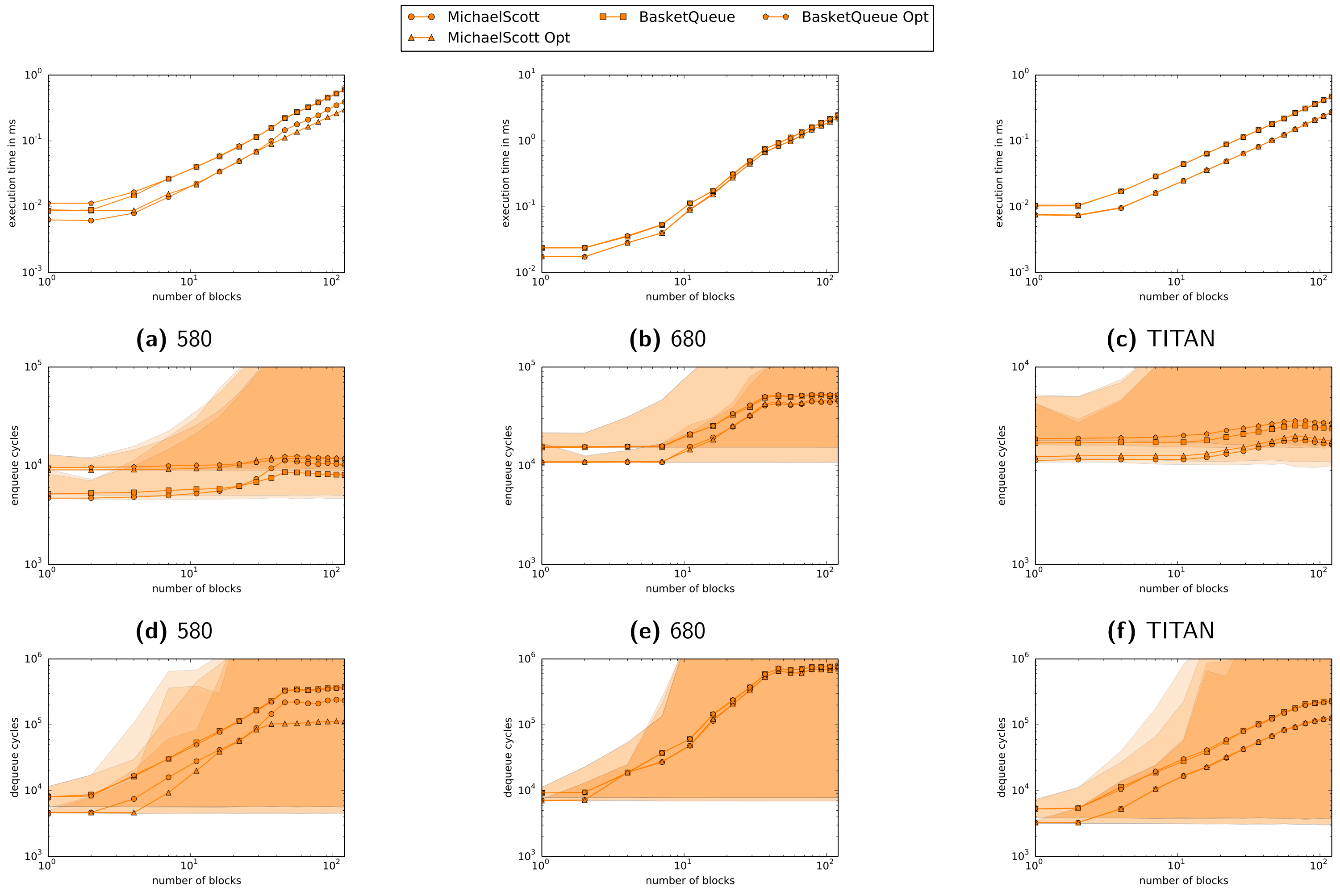


(a) 580 **(b)** 680 **(c)** TITAN

(d) 580 **(e)** 680 **(f)** TITAN

(g) 580 **(h)** 680 **(i)** TITAN

Figure C.10: Linked Queue Package Scenario

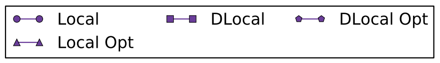


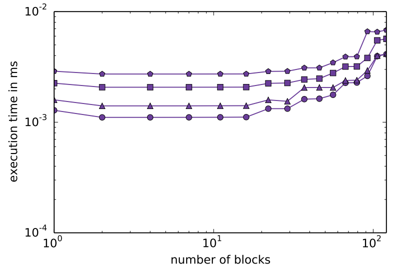
(a) 580 **(b)** 680 **(c)** TITAN

(d) 580 **(e)** 680 **(f)** TITAN

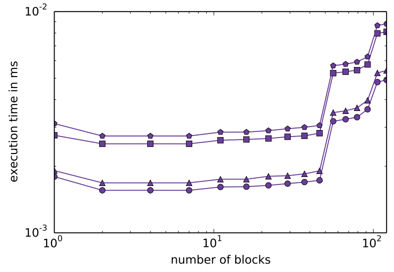
(g) 580 **(h)** 680 **(i)** TITAN

Figure C.11: Michael Scott Package Scenario

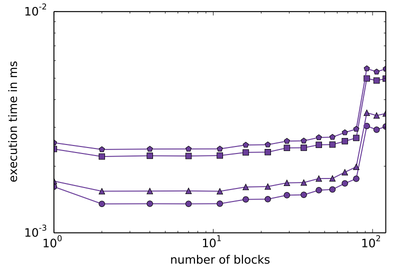




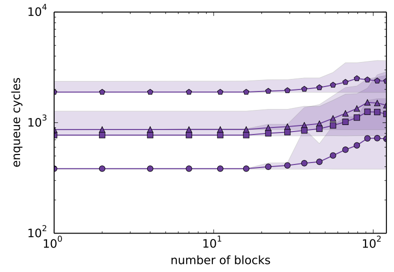
(a) 580



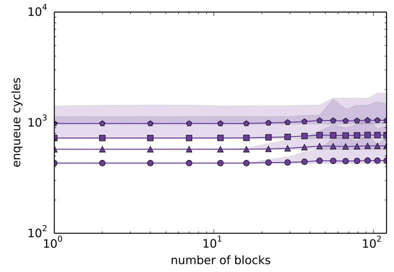
(b) 680



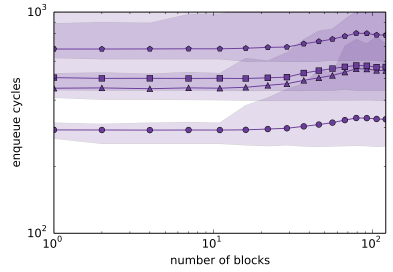
(c) TITAN



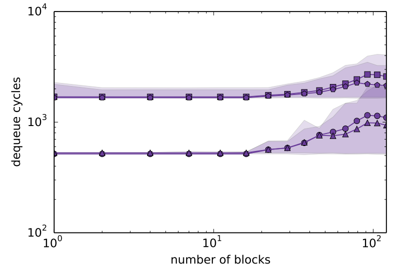
(d) 580



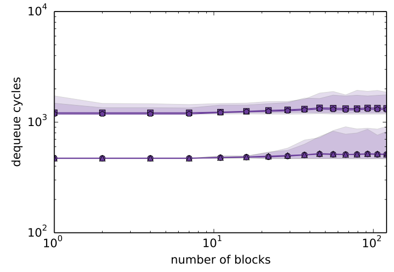
(e) 680



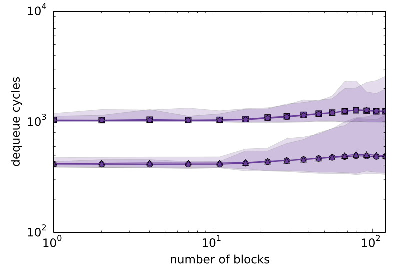
(f) TITAN



(g) 580

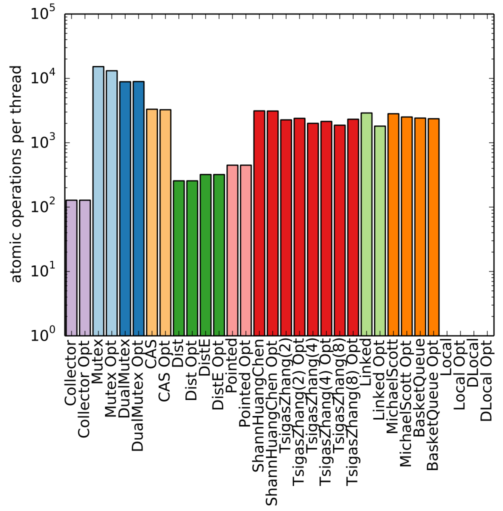


(h) 680

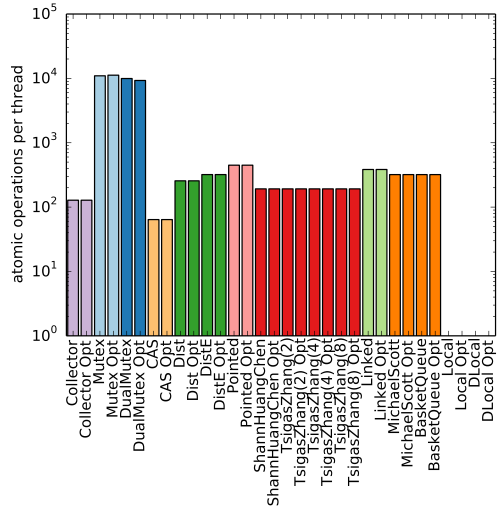


(i) TITAN

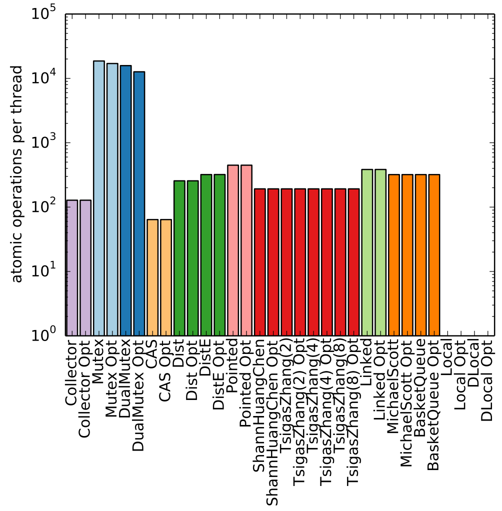
Figure C.12: Local Queues Package Scenario



(a) 580

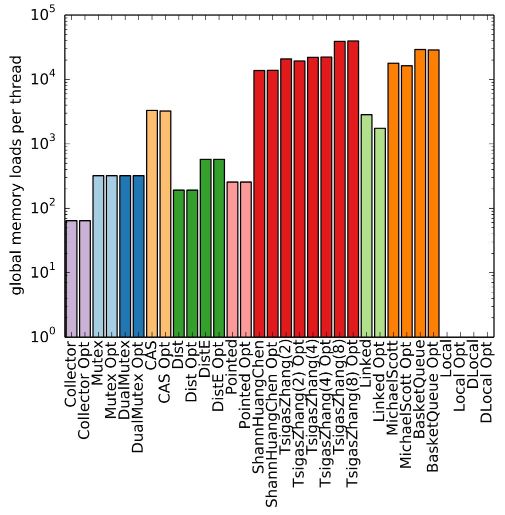


(b) 680

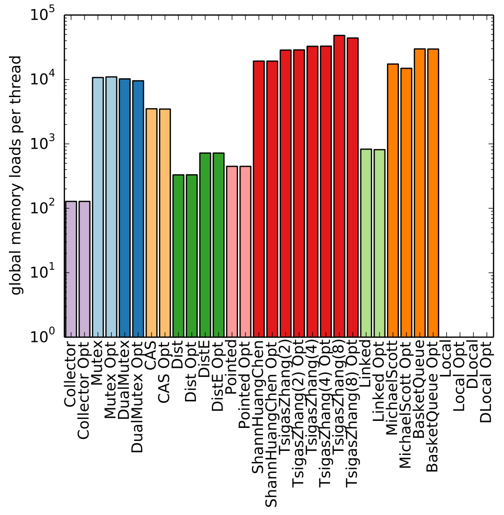


(c) TITAN

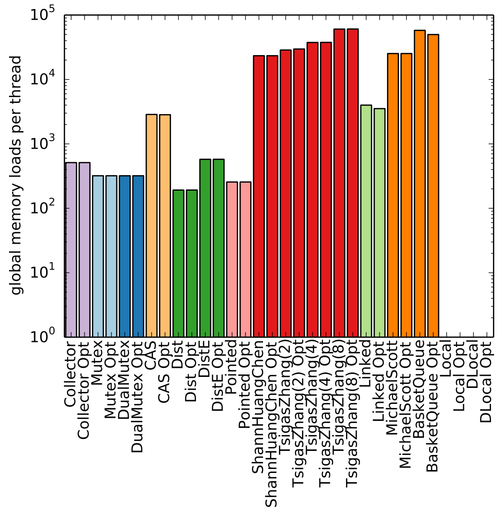
Figure C.13: Atomic Operations Package Scenario



(a) 580

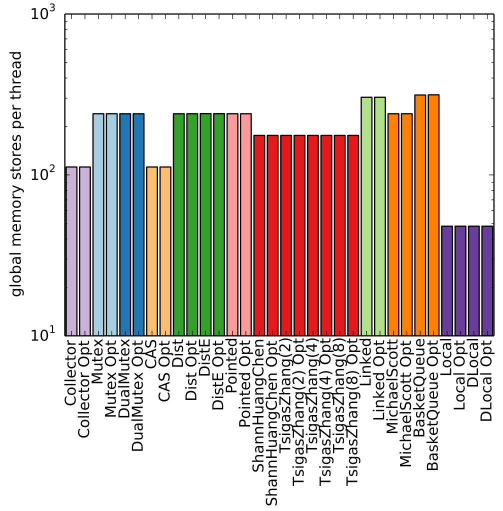


(b) 680

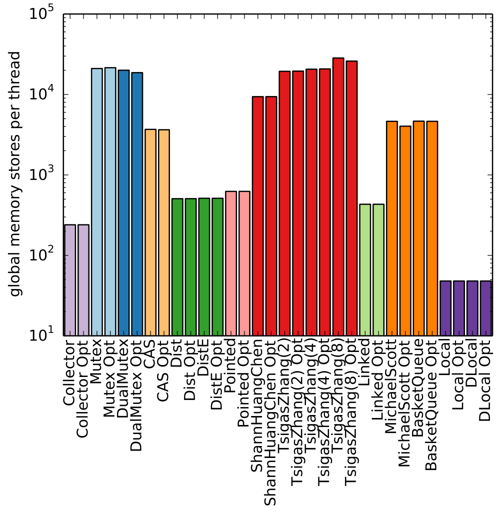


(c) TITAN

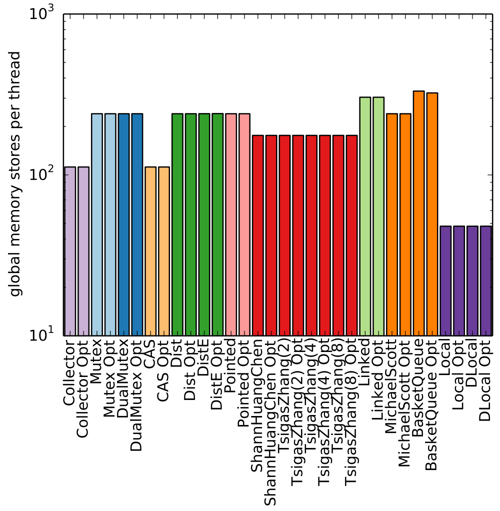
Figure C.14: Global Loads Package Scenario



(a) 580

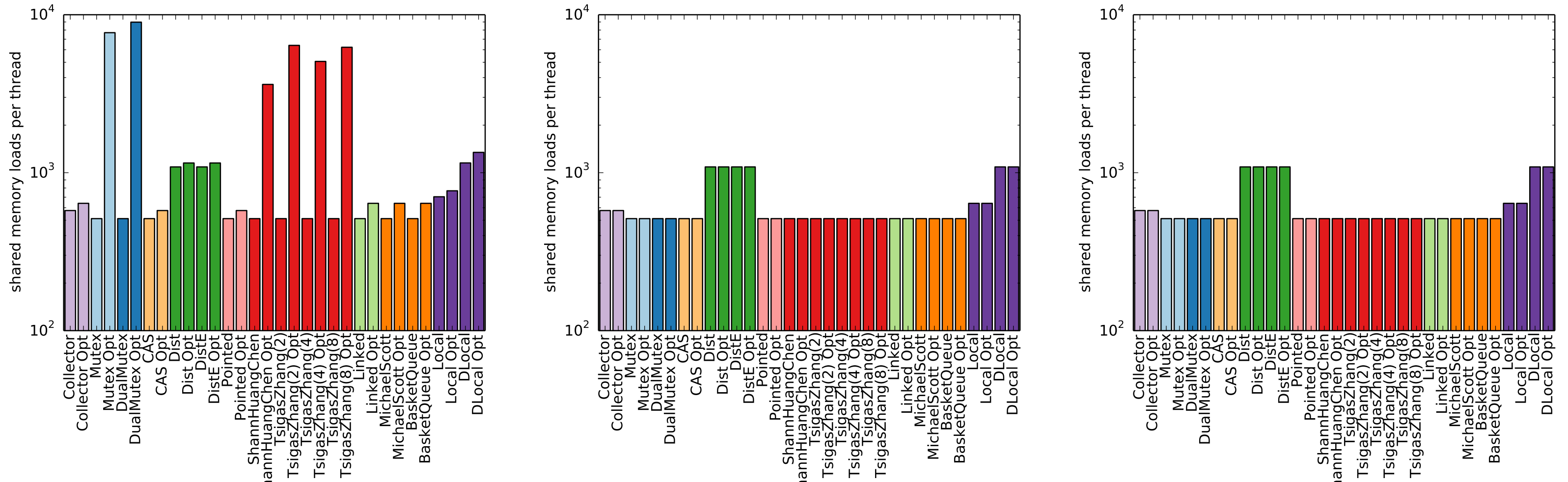


(b) 680



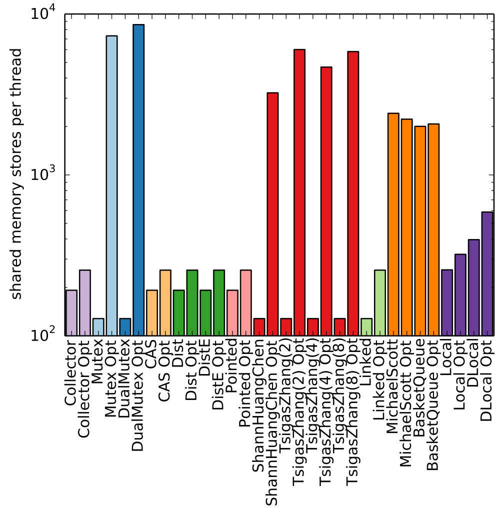
(c) TITAN

Figure C.15: Global Stores Package Scenario

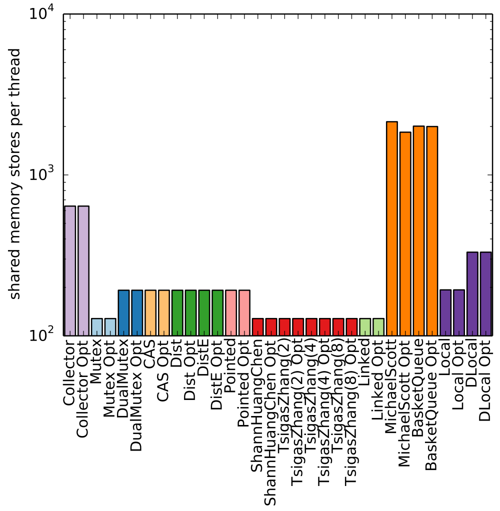


(a) 580 **(b)** 680 **(c)** TITAN

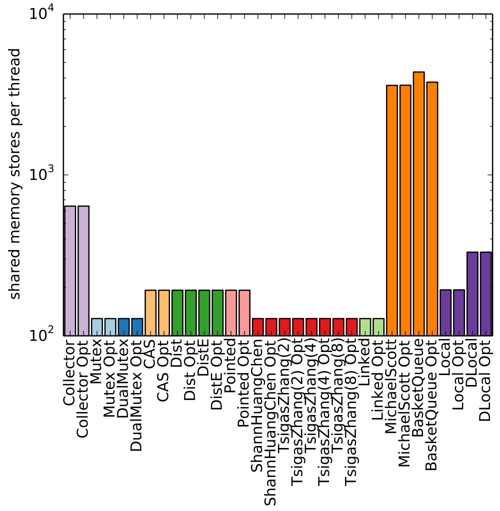
Figure C.16: Shared Loads Package Scenario



(a) 580



(b) 680



(c) TITAN

Figure C.17: Shared Stores Package Scenario

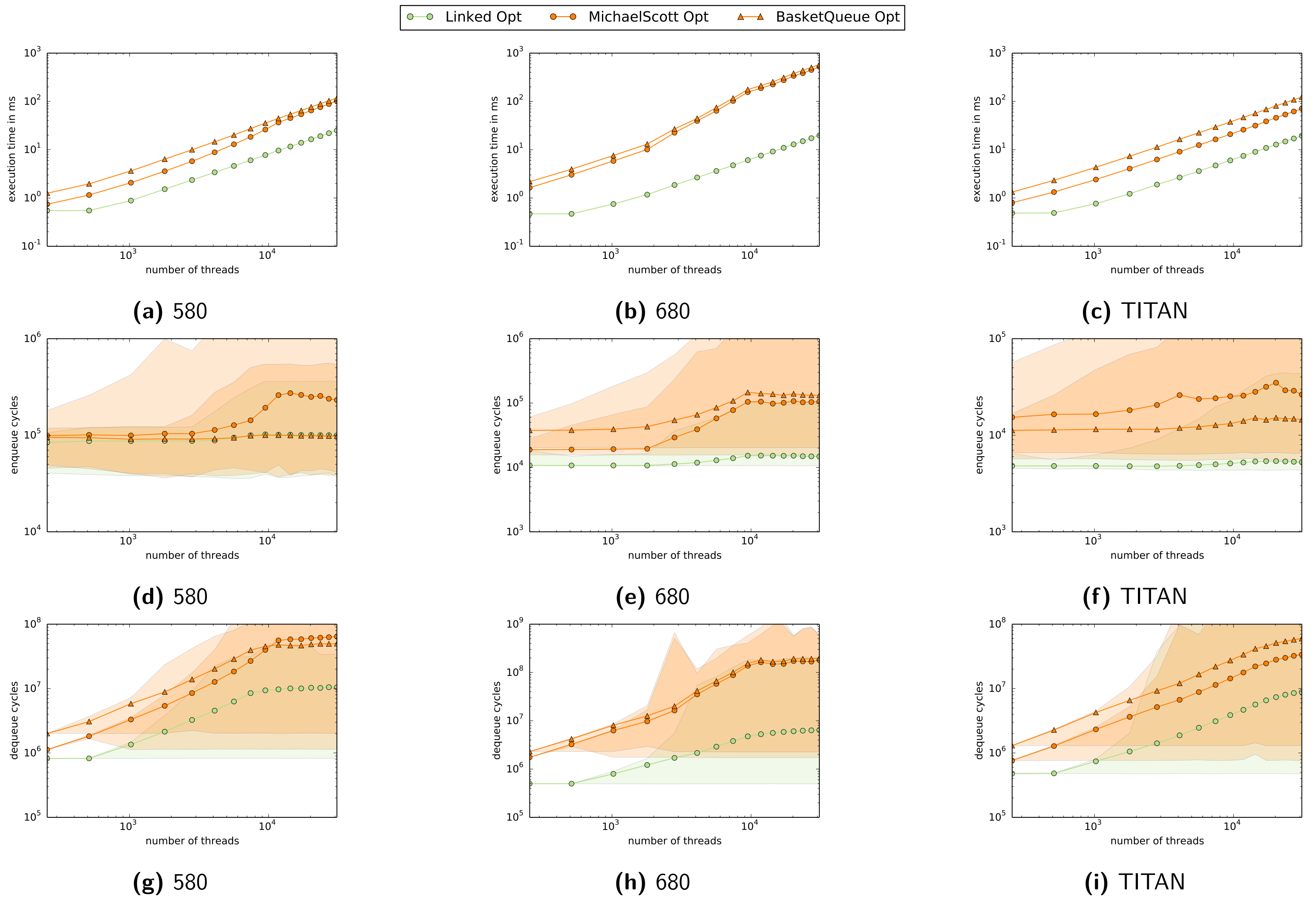
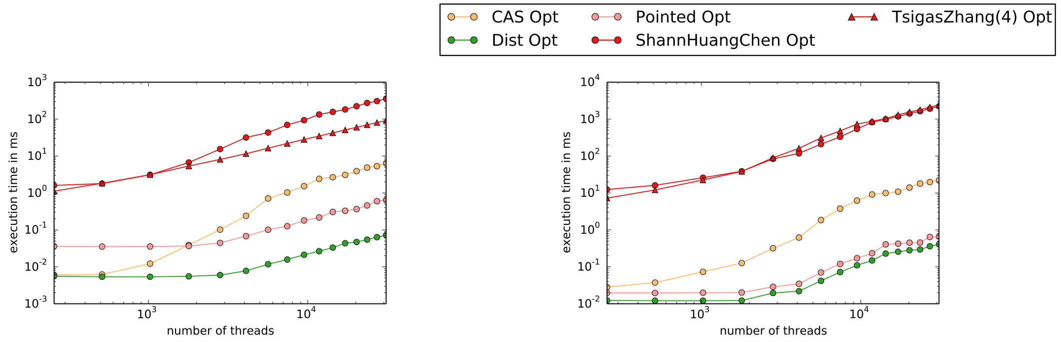
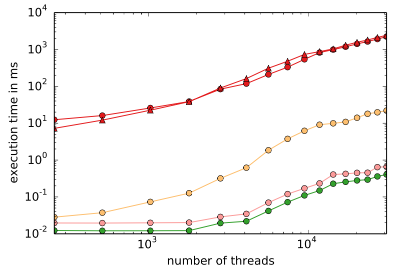


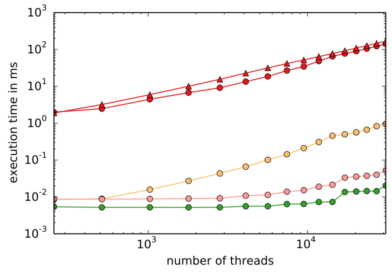
Figure C.18: Linked Queues Item Scenario



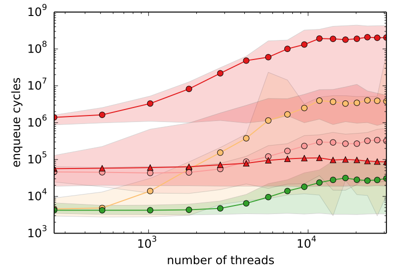
(a) 580



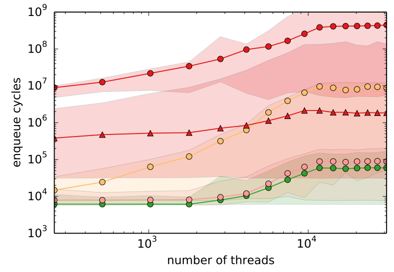
(b) 680



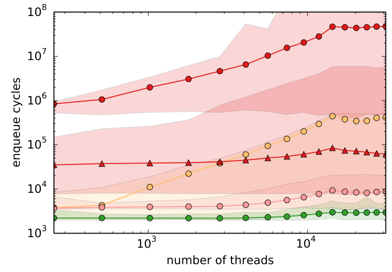
(c) TITAN



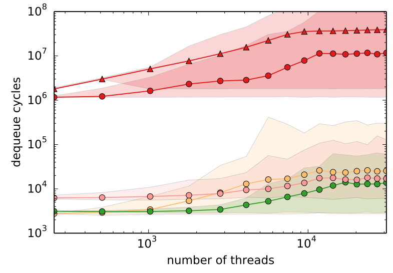
(d) 580



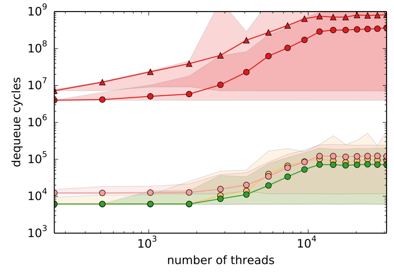
(e) 680



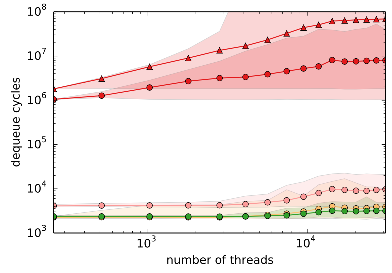
(f) TITAN



(g) 580

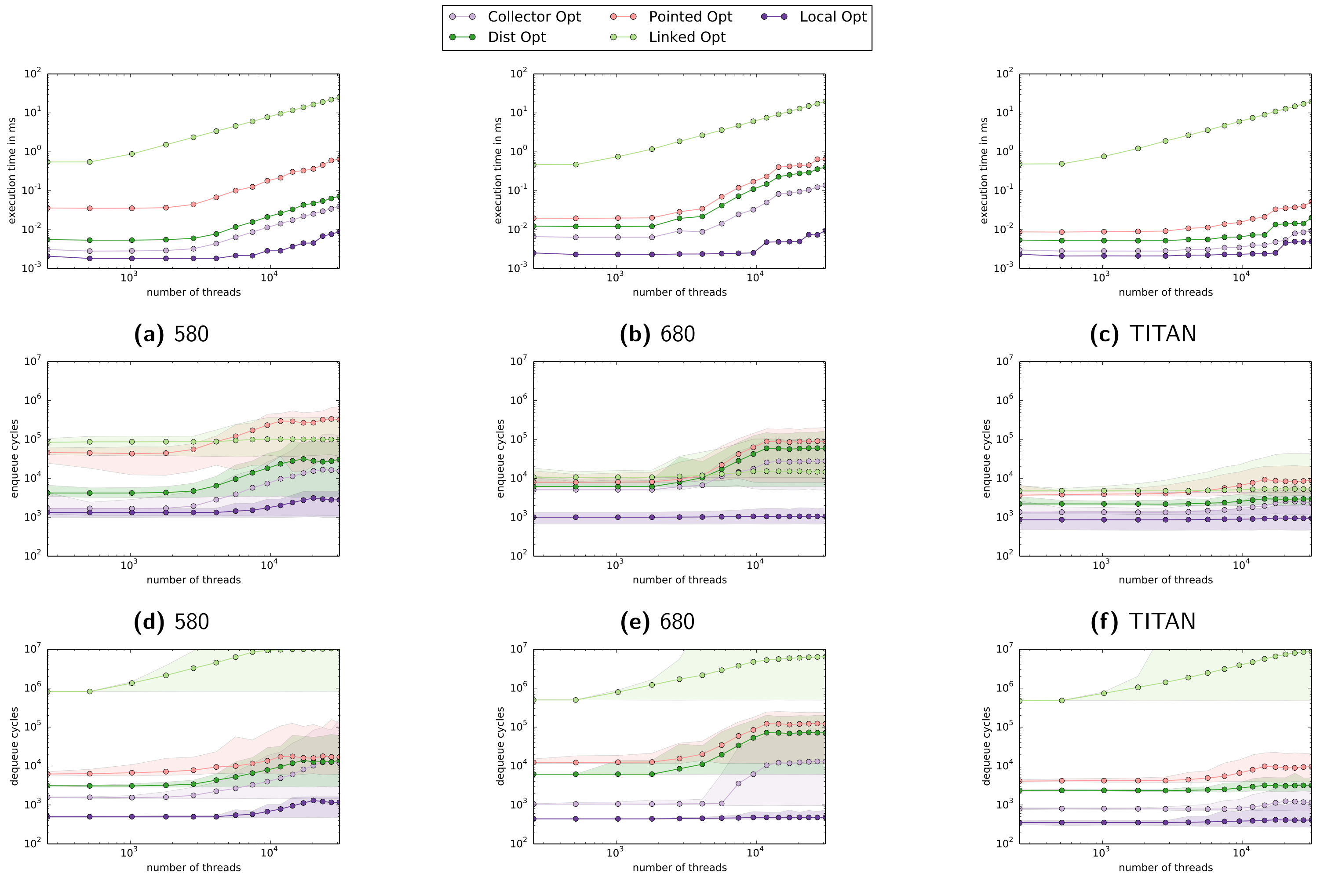


(h) 680



(i) TITAN

Figure C.19: Array Queues Item Scenario

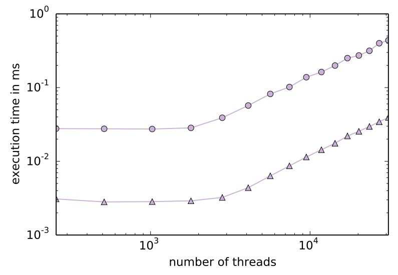


(a) 580 **(b)** 680 **(c)** TITAN

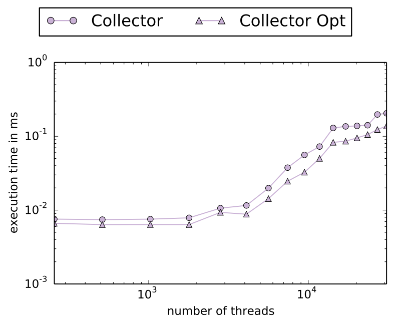
(d) 580 **(e)** 680 **(f)** TITAN

(g) 580 **(h)** 680 **(i)** TITAN

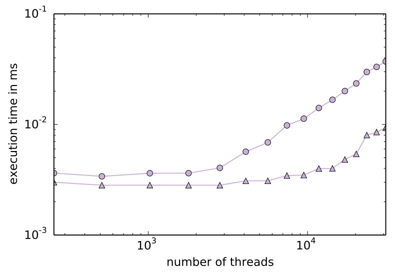
Figure C.20: Overview Item Scenario



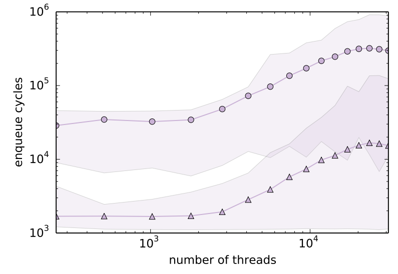
(a) 580



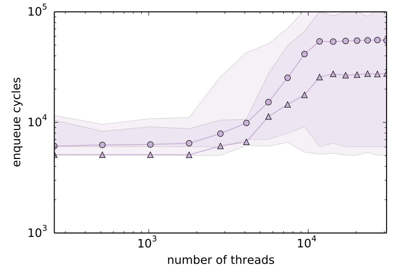
(b) 680



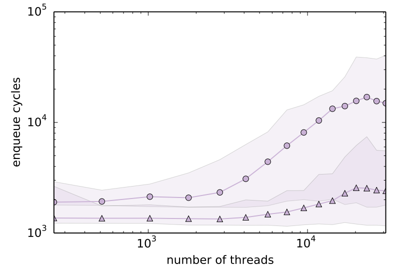
(c) TITAN



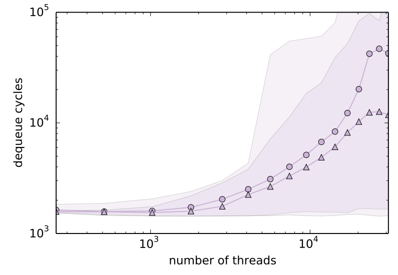
(d) 580



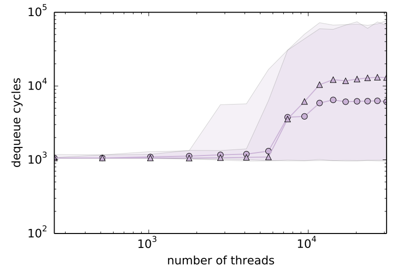
(e) 680



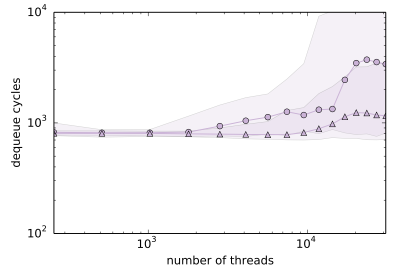
(f) TITAN



(g) 580

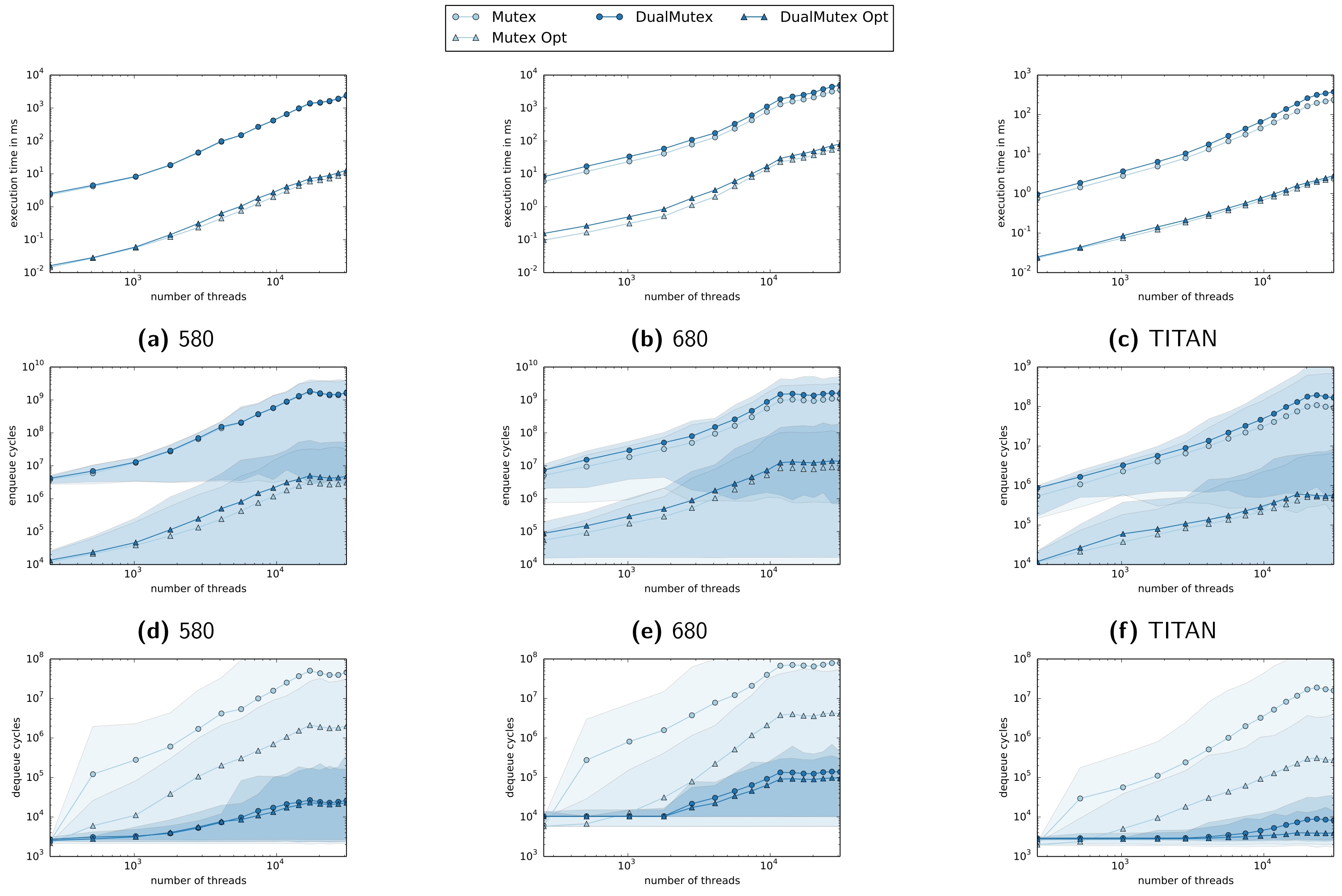


(h) 680



(i) TITAN

Figure C.21: Collector Queue Item Scenario

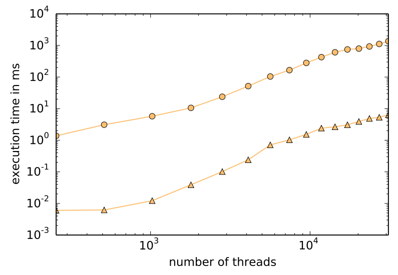


(a) 580 **(b)** 680 **(c)** TITAN

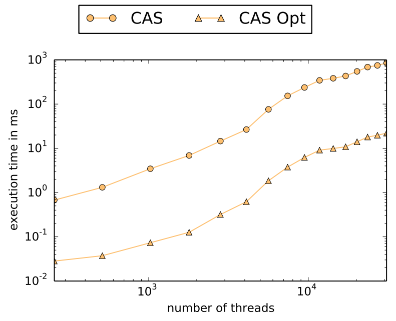
(d) 580 **(e)** 680 **(f)** TITAN

(g) 580 **(h)** 680 **(i)** TITAN

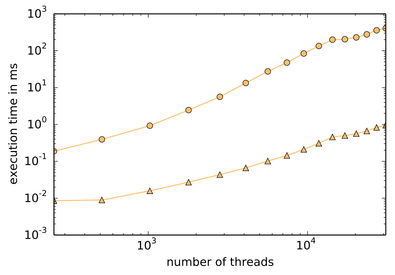
Figure C.22: Mutex Queue Item Scenario



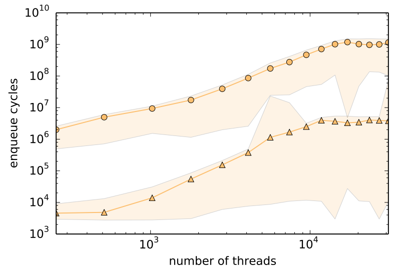
(a) 580



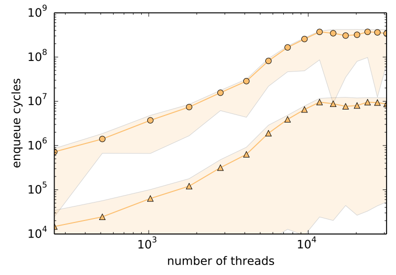
(b) 680



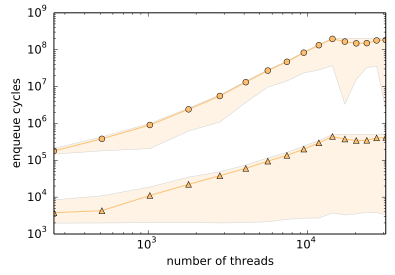
(c) TITAN



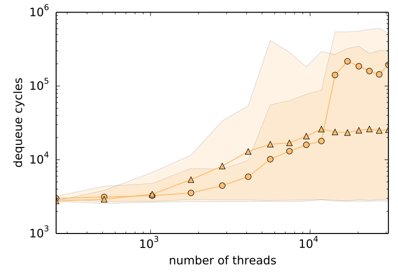
(d) 580



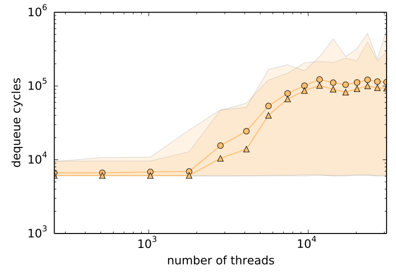
(e) 680



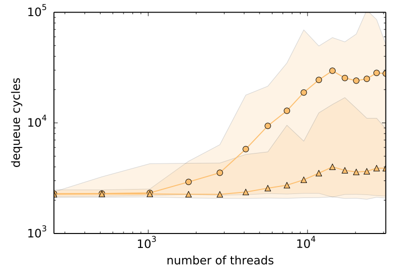
(f) TITAN



(g) 580

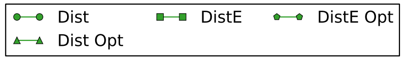


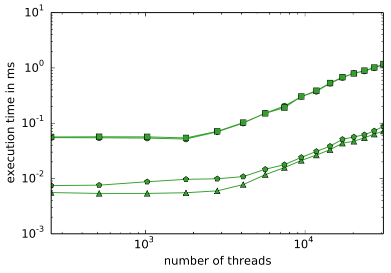
(h) 680



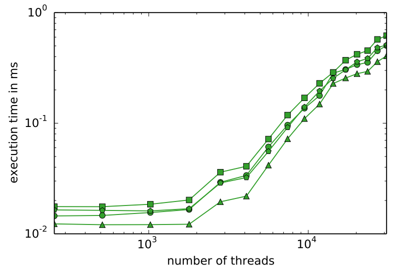
(i) TITAN

Figure C.23: CAS Ordered Queue Item Scenario

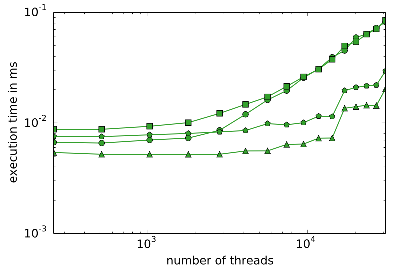




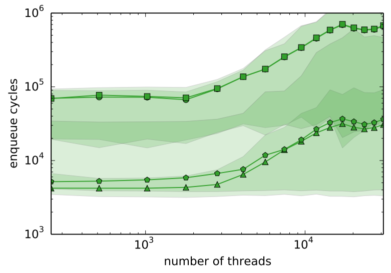
(a) 580



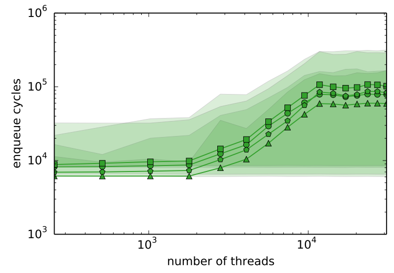
(b) 680



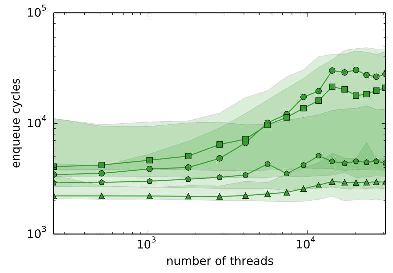
(c) TITAN



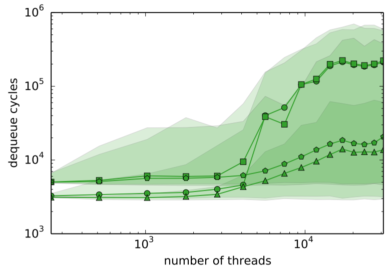
(d) 580



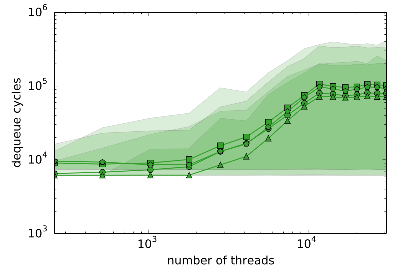
(e) 680



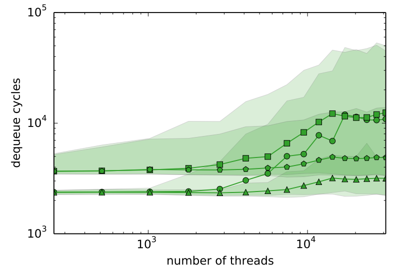
(f) TITAN



(g) 580

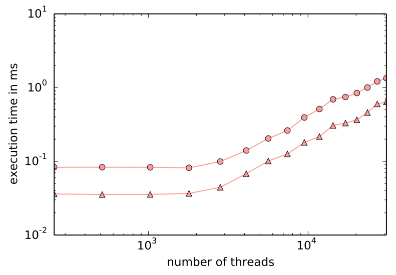


(h) 680

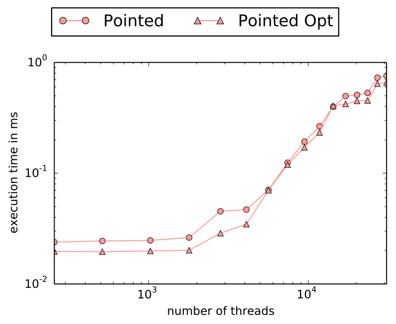


(i) TITAN

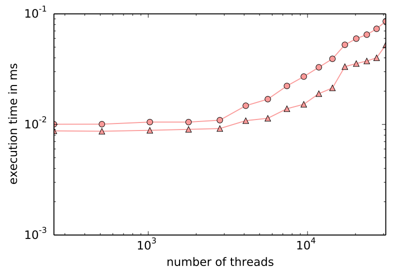
Figure C.24: Distributed Locks Queue Item Scenario



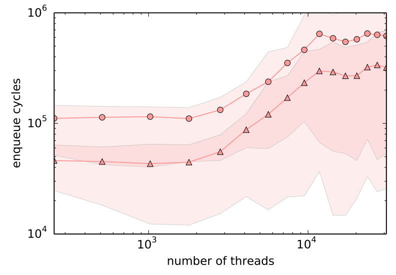
(a) 580



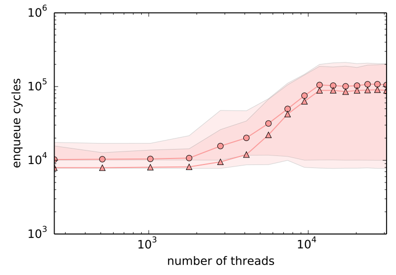
(b) 680



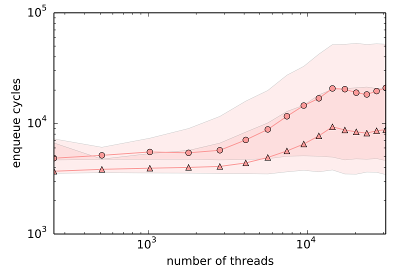
(c) TITAN



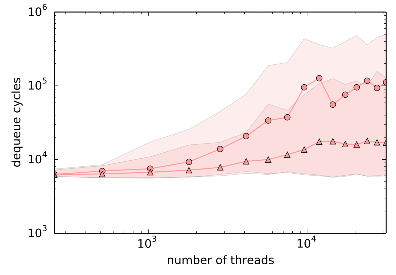
(d) 580



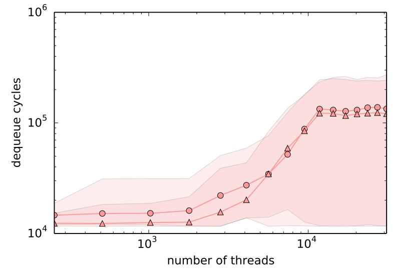
(e) 680



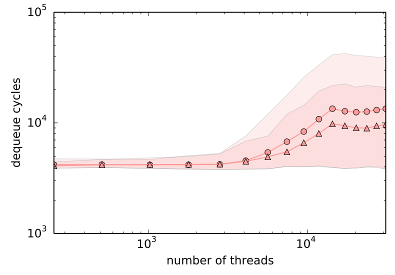
(f) TITAN



(g) 580

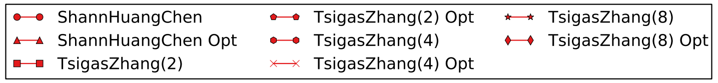


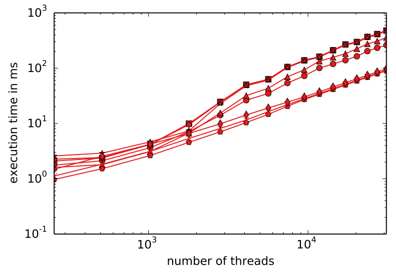
(h) 680



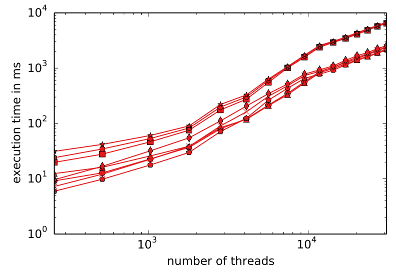
(i) TITAN

Figure C.25: Pointed Queue Item Scenario

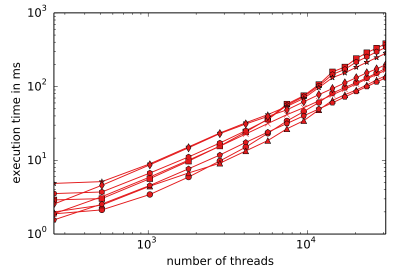




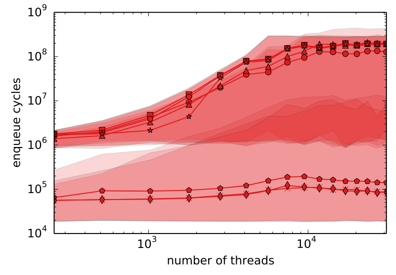
(a) 580



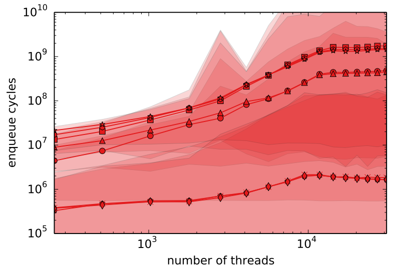
(b) 680



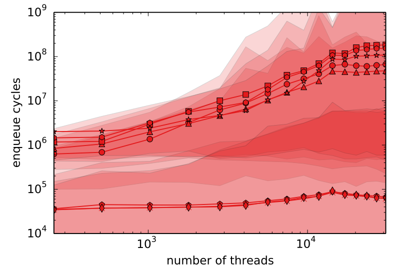
(c) TITAN



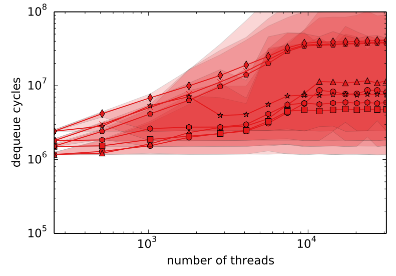
(d) 580



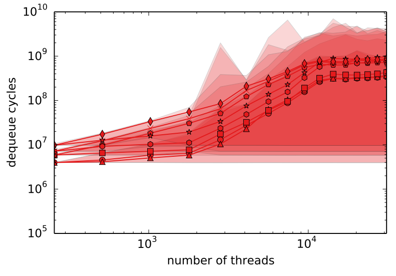
(e) 680



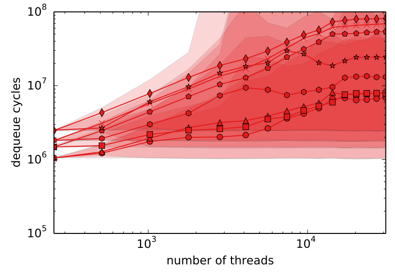
(f) TITAN



(g) 580

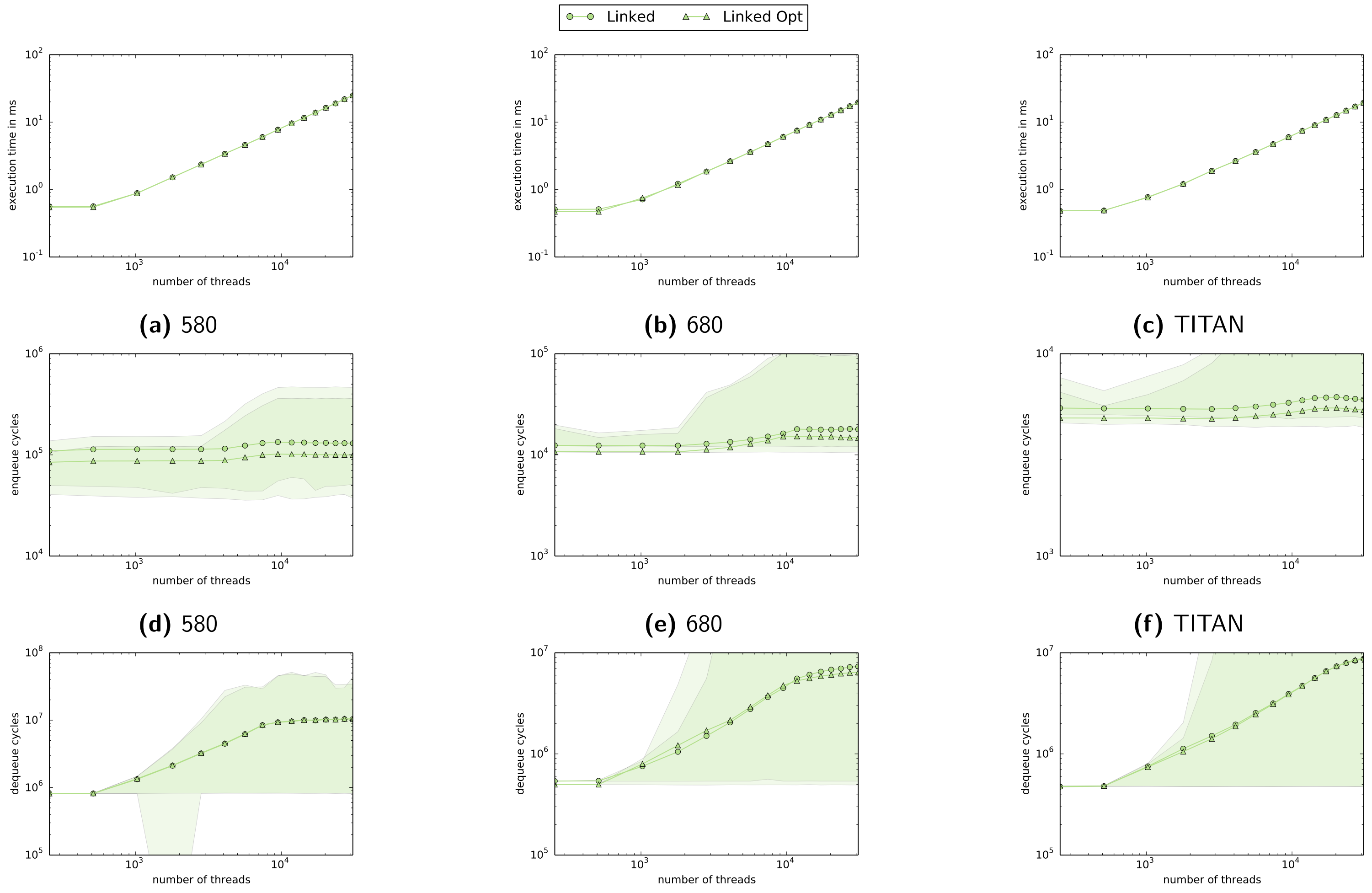


(h) 680



(i) TITAN

Figure C.26: Shan Huang Chen Item Scenario

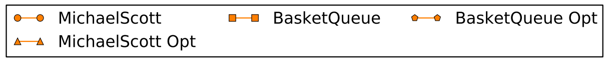


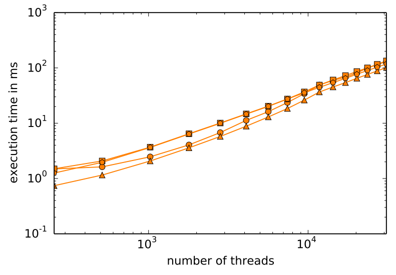
(a) 580 **(b)** 680 **(c)** TITAN

(d) 580 **(e)** 680 **(f)** TITAN

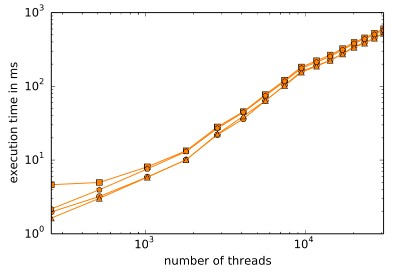
(g) 580 **(h)** 680 **(i)** TITAN

Figure C.27: Linked Queue Item Scenario

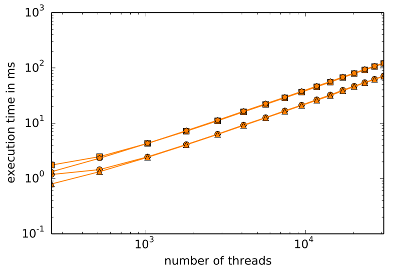




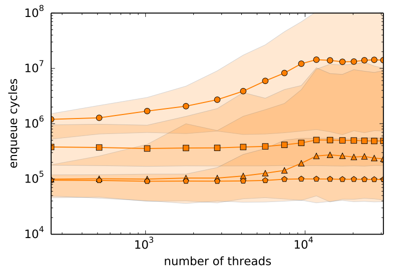
(a) 580



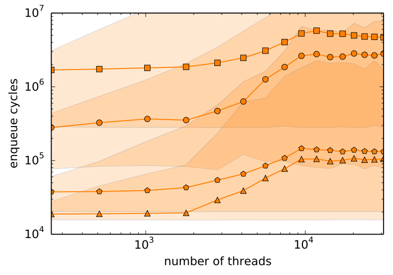
(b) 680



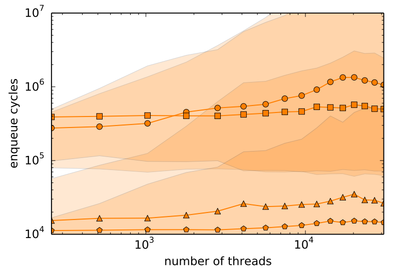
(c) TITAN



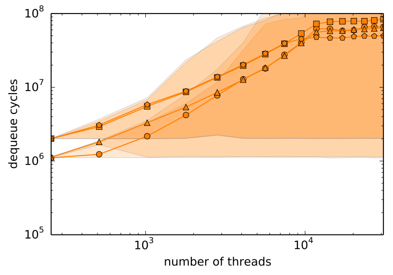
(d) 580



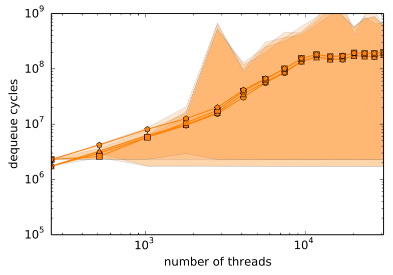
(e) 680



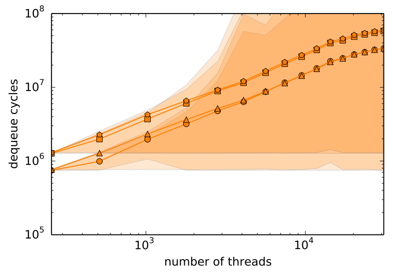
(f) TITAN



(g) 580

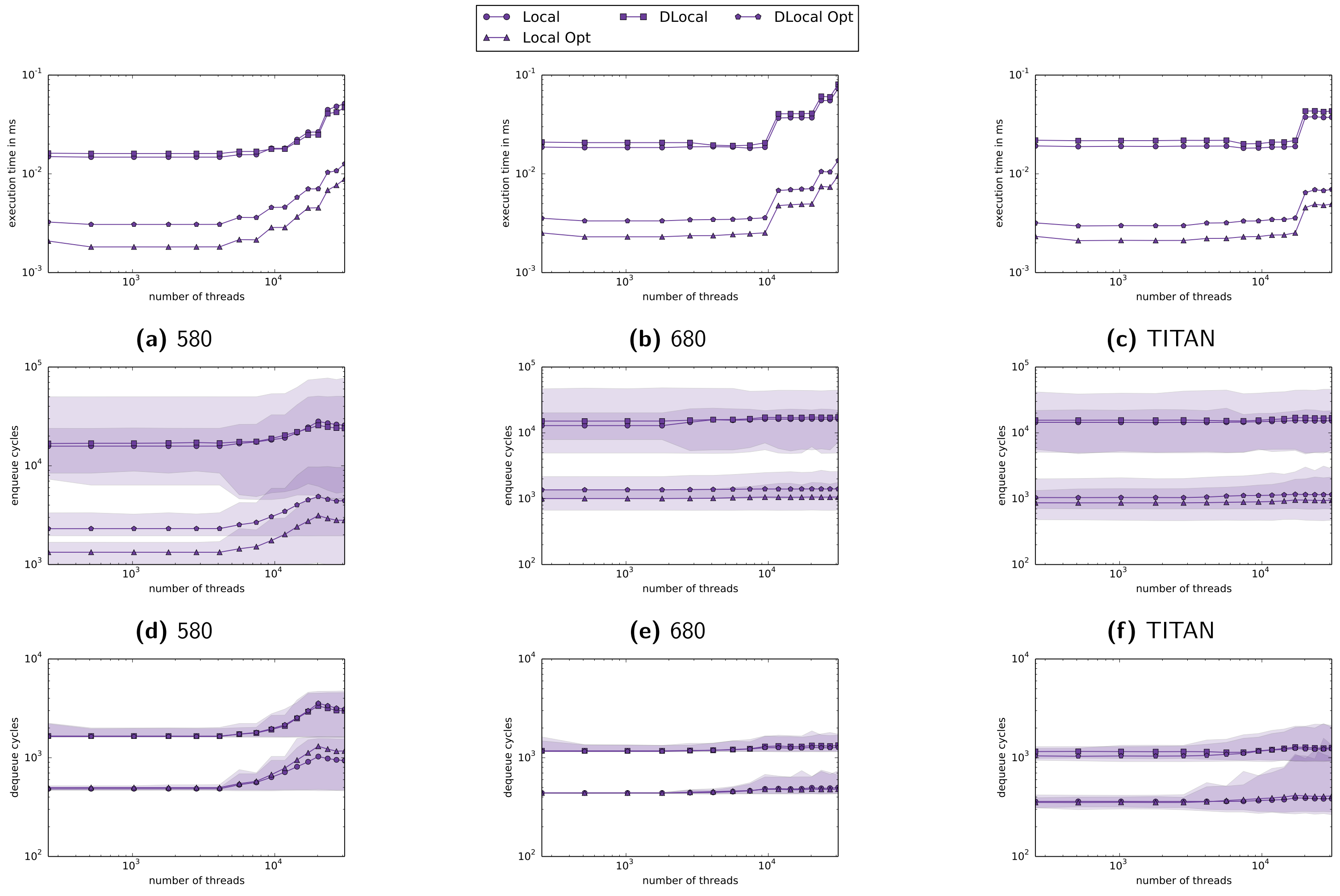


(h) 680



(i) TITAN

Figure C.28: Michael Scott Item Scenario

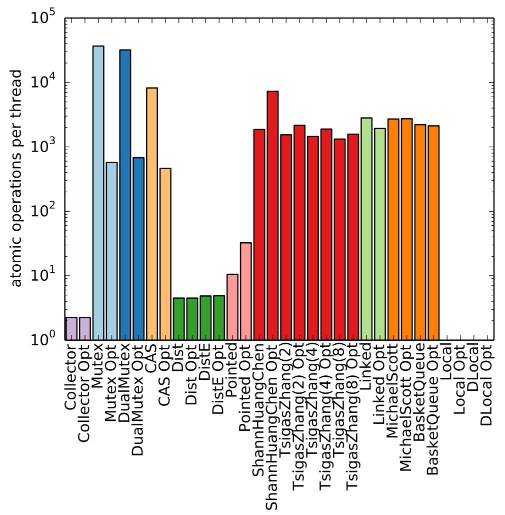


(a) 580 **(b)** 680 **(c)** TITAN

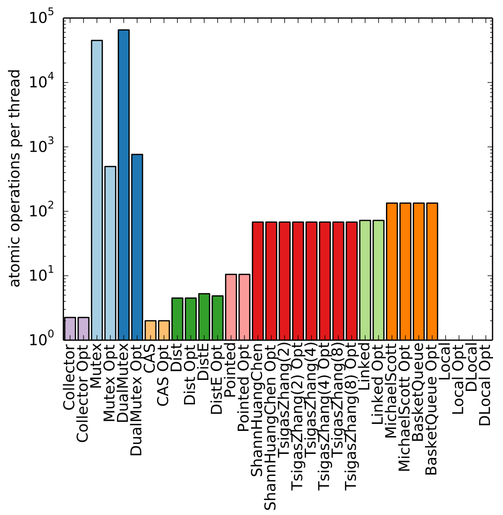
(d) 580 **(e)** 680 **(f)** TITAN

(g) 580 **(h)** 680 **(i)** TITAN

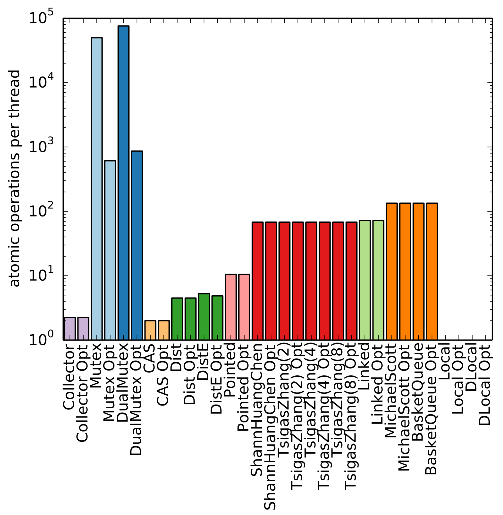
Figure C.29: Local Queues Item Scenario



(a) 580

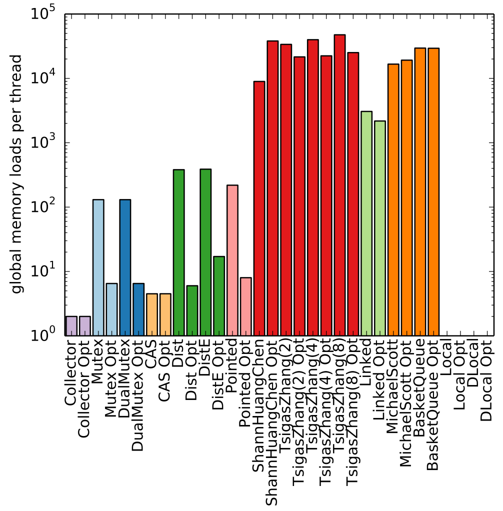


(b) 680

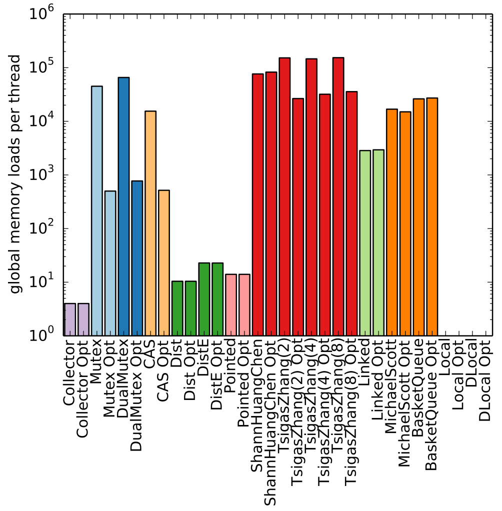


(c) TITAN

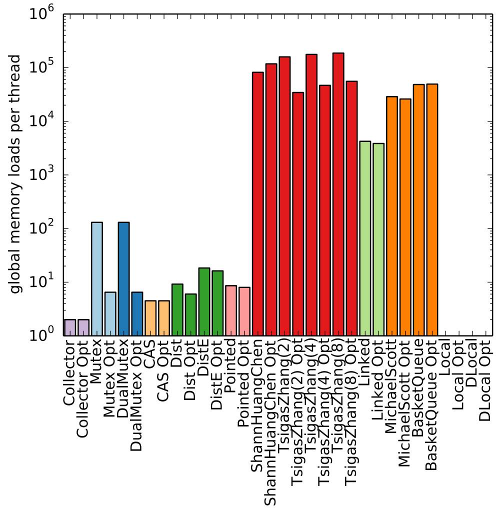
Figure C.30: Atomic Operations Item Scenario



(a) 580

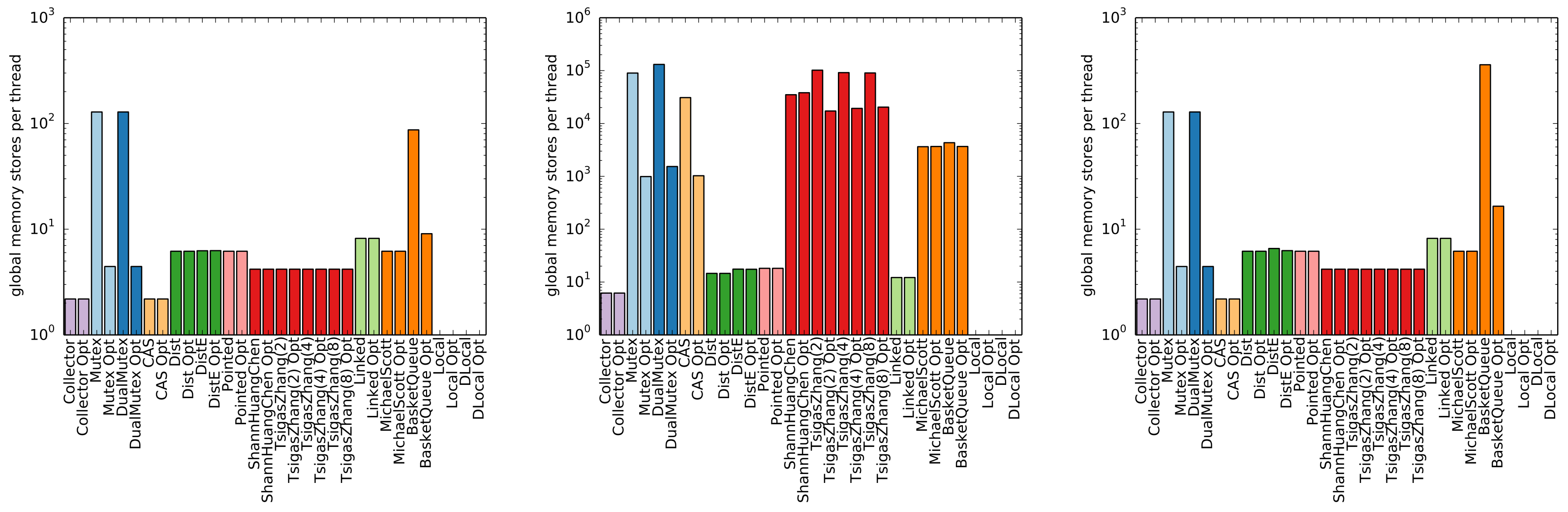


(b) 680



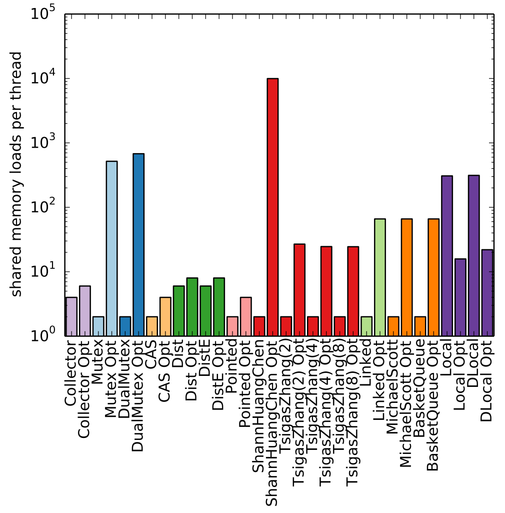
(c) TITAN

Figure C.31: Global Loads Item Scenario

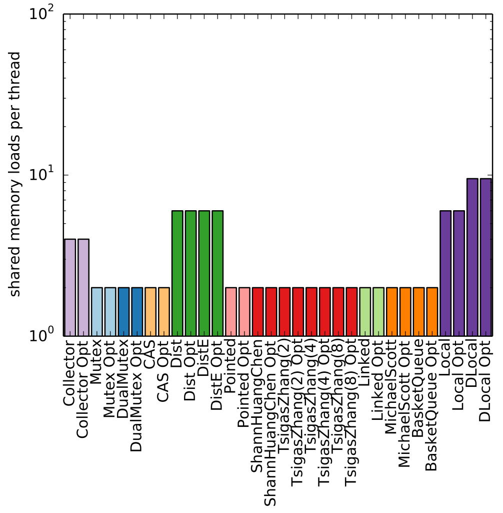


(a) 580 **(b)** 680 **(c)** TITAN

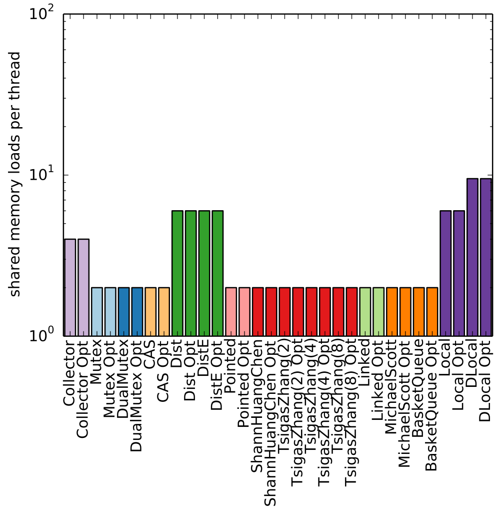
Figure C.32: Global Stores Item Scenario



(a) 580

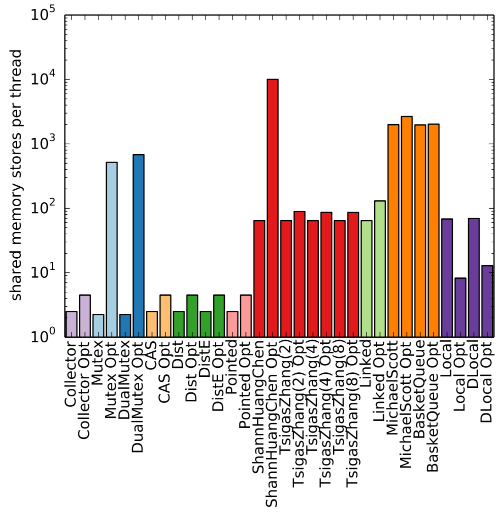


(b) 680

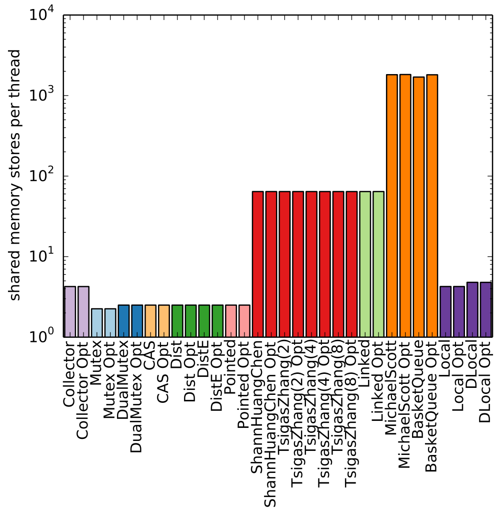


(c) TITAN

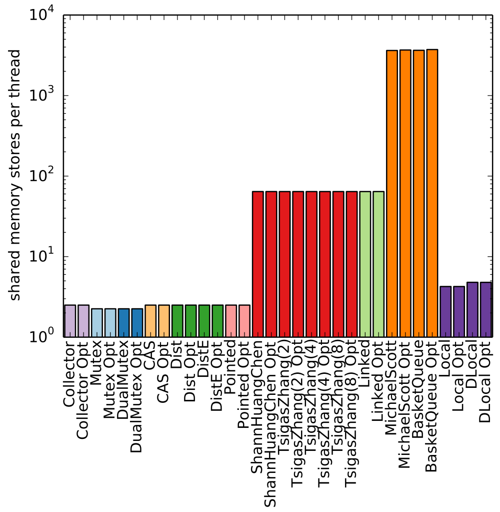
Figure C.33: Shared Loads Item Scenario



(a) 580



(b) 680



(c) TITAN

Figure C.34: Shared Stores Item Scenario

Appendix D

Pipeline Evaluation in Detail

	4 Stages															
	Big								Small							
	Mixed				Packages				Mixed				Packages			
	et		vt		et		vt		et		vt		et		vt	
	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr
K	89.1	66.8	192	148	7.4	5.6	**12.9**	10.9	97.4	74.9	201	146	7.2	6.3	12.8	**10.4**
KwQ	54.0	56.0	**101**	**118**	10.7	5.6	12.9	**10.3**	59.5	58.5	113	117	7.2	5.6	**12.8**	11.8
KwQ_A	58.3	63.0	142	168	9.0	5.7	15.9	13.0	64.6	58.6	152	161	7.0	5.6	16.1	12.9
KwQ_{A+S}	60.1	59.8	157	162	6.9	5.6	15.9	12.9	68.8	59.3	162	159	7.1	5.6	16.1	14.0
VMK_B	51.1	49.9	200	179	7.6	6.0	39.9	30.1	46.4	46.1	186	174	7.6	5.8	38.9	29.9
$VMK_{B,shared}$	**38.5**	37.5	170	164	**6.5**	4.4	27.4	17.3	29.6	27.2	98.5	119	6.7	**4.4**	26.6	15.8
VMK	51.1	50.4	194	173	7.5	6.1	21.1	17.0	44.2	45.8	182	179	7.6	5.9	20.9	16.8
VMK_{shared}	39.1	**37.5**	164	162	6.6	**4.4**	13.4	10.4	28.7	27.7	101	118	6.6	4.5	13.9	10.6
$VMK_{shared,out}$	38.5	37.6	168	162	6.6	4.4	13.4	10.4	**26.6**	**26.0**	**74.3**	**69.5**	**6.6**	4.5	13.9	10.7
$VMK_{shared,dyn}$	67.4	65.7	269	282	8.3	6.0	22.2	17.4	66.0	64.4	179	177	8.5	6.1	21.8	17.0

Table D.1: Pipeline test with synthetic FMA load on a GTX 580

	7 Stages															
	Big								Small							
	Mixed				Packages				Mixed				Packages			
	et		vt		et		vt		et		vt		et		vt	
	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr
K	246	189	286	229	19.7	16.7	**127**	108	262	202	284	254	18.2	15.9	122	**107**
KwQ	98.0	98.2	**132**	**169**	18.5	16.3	122	**108**	105	97.6	143	167	18.2	16.1	**121**	106
KwQ_A	96.4	105	196	207	17.8	16.0	124	123	87.0	88.0	184	200	17.5	15.8	124	115
KwQ_{A+S}	89.8	104	205	212	17.8	15.9	124	112	93.7	89.9	187	204	18.4	15.9	123	110
VMK_B	80.6	82.3	250	242	19.5	17.6	241	241	80.7	80.9	248	240	18.7	16.9	238	233
$VMK_{B,shared}$	**72.9**	71.6	238	220	**16.4**	13.5	166	161	56.6	49.8	158	130	16.2	**13.3**	140	159
VMK	83.6	76.5	231	207	19.2	17.9	158	144	81.0	76.1	217	199	18.9	17.5	157	143
VMK_{shared}	67.9	**75.6**	216	200	16.2	**13.6**	132	110	53.0	53.0	141	151	16.3	13.5	131	99.8
$VMK_{shared,out}$	76.8	76.0	225	211	16.3	13.6	132	112	**49.0**	**47.7**	**183**	**168**	**16.2**	13.6	132	97.4
$VMK_{shared,dyn}$	130	131	418	408	21.1	18.8	196	179	90.7	90.0	266	316	20.6	18.7	188	168

Table D.2: Pipeline test with synthetic FMA load on a GTX 580

	4 Stages															
	Big								Small							
	Mixed				Packages				Mixed				Packages			
	et		vt		et		vt		et		vt		et		vt	
	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr
K	414	352	1068	1066	14.8	13.6	52.1	47.0	314	296	901	1089	15.9	13.0	53.9	47.7
KwQ	281	330	941	1074	16.3	14.6	53.7	50.7	341	337	816	934	14.6	13.8	51.2	49.1
KwQ_A	372	316	996	981	15.5	13.8	52.7	48.8	338	313	805	791	14.2	13.5	52.3	48.6
KwQ_{A+S}	389	319	1028	974	15.5	13.9	52.8	48.8	330	309	790	791	14.3	13.6	51.1	48.6
VMK_B	362	308	870	879	32.0	26.9	117	106	363	284	903	795	32.2	27.3	115	104
$VMK_{B,shared}$	197	207	837	794	**5.2**	**3.4**	19.3	17.4	99.2	110	430	302	5.3	**3.3**	18.6	17.0
VMK	346	331	895	**779**	31.7	27.0	67.3	59.0	370	293	873	781	34.4	26.6	67.4	58.5
VMK_{shared}	198	200	835	835	5.4	3.4	14.3	**10.6**	95.6	98.7	471	570	**4.8**	3.5	**13.9**	**10.8**
$VMK_{shared,out}$	**178**	**175**	**811**	824	5.3	3.4	**14.1**	10.9	**60.1**	**63.0**	**227**	**292**	4.9	3.4	14.5	11.2
$VMK_{shared,dyn}$	183	201	922	845	7.2	5.3	28.6	25.1	272	280	678	707	6.7	5.2	28.5	24.4

Table D.3: Pipeline test with synthetic FMA load on a GTX 680

	7 Stages															
	Big								Small							
	Mixed				Packages				Mixed				Packages			
	et		vt		et		vt		et		vt		et		vt	
	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr
K	1175	1013	1417	1394	34.4	33.2	331	312	589	776	1136	1368	34.6	33.0	325	303
KwQ	410	783	1181	1292	38.1	36.4	239	227	684	730	1017	1078	37.1	34.3	232	222
KwQ_A	687	553	1232	1183	35.8	34.9	235	218	785	642	977	957	35.6	33.9	232	215
KwQ_{A+S}	658	562	1235	1150	35.8	34.9	237	218	699	649	1026	955	35.6	33.6	232	215
VMK_B	602	567	1197	1093	85.3	73.1	676	671	633	534	1143	1143	86.0	69.4	673	657
$VMK_{B,shared}$	413	427	1124	1014	**12.4**	**9.7**	154	145	269	304	736	686	11.9	**9.7**	154	143
VMK	620	539	1083	**1064**	81.0	70.9	369	353	621	550	977	1127	75.5	73.2	365	350
VMK_{shared}	433	403	1116	996	12.4	9.5	91.7	**82.6**	300	289	705	734	**12.0**	9.6	**91.4**	**78.2**
$VMK_{shared,out}$	**416**	**438**	**1079**	1074	12.1	9.4	**97.9**	83.4	**258**	**299**	**837**	**737**	12.4	10.6	92.9	79.1
$VMK_{shared,dyn}$	406	424	1153	991	21.4	16.5	270	250	363	347	1141	708	16.1	16.2	264	253

Table D.4: Pipeline test with synthetic FMA load on a GTX 680

	4 Stages															
	Big								Small							
	Mixed				Packages				Mixed				Packages			
	et		vt		et		vt		et		vt		et		vt	
	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr
K	48.2	39.2	138	120	6.3	5.3	12.4	10.7	47.9	39.4	134	117	6.2	4.9	10.5	8.5
KwQ	27.4	27.3	99.4	**96.4**	6.4	5.3	12.4	10.7	25.4	26.1	96.1	93.8	6.2	5.0	10.6	8.5
KwQ_A	**22.6**	28.8	**85.7**	103	6.1	5.0	11.6	10.2	22.2	27.6	83.8	102	6.1	4.8	10.4	8.7
KwQ_{A+S}	23.1	27.6	86.9	106	6.1	5.0	11.6	10.2	22.3	26.4	83.8	97.6	6.2	4.8	10.4	8.7
DP	n.a.	n.a.	n.a.	n.a.	114	101	336	308	n.a.	n.a.	n.a.	n.a.	110	99.7	332	305
HDP	37.7	32.8	196	139	5.6	4.0	11.0	8.9	39.5	29.0	198	134	11.3	3.7	10.5	8.0
HDP_A	44.3	39.6	230	180	5.2	3.6	11.0	8.6	45.4	37.2	224	177	5.8	3.7	10.9	8.4
VMK_B	39.7	39.0	125	120	4.8	3.9	26.5	19.4	31.4	29.5	130	113	**5.0**	3.9	25.0	19.9
$VMK_{B,shared}$	28.1	27.8	123	118	4.8	**3.4**	19.3	11.4	22.9	20.0	70.2	76.3	5.3	**3.4**	18.0	11.5
VMK	40.9	36.8	132	125	5.8	4.5	15.7	13.2	30.2	28.6	129	109	5.5	4.6	12.9	10.7
VMK_{shared}	27.6	27.8	121	122	**4.8**	3.5	8.9	6.5	21.8	21.0	76.1	76.2	5.3	3.4	8.7	6.4
$VMK_{shared,out}$	27.6	**26.8**	115	118	5.0	3.4	**8.9**	**6.4**	**17.2**	**16.7**	**66.9**	**61.5**	5.3	3.4	**8.6**	**6.3**
$VMK_{shared,dyn}$	44.3	43.3	160	160	5.0	3.6	12.7	12.3	43.5	47.8	124	118	5.7	4.1	12.6	10.1

Table D.5: Pipeline test with synthetic FMA load on a GTX TITAN

	7 Stages															
	Big								Small							
	Mixed				Packages				Mixed				Packages			
	et		vt		et		vt		et		vt		et		vt	
	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr
K	126	110	174	148	14.7	12.4	85.4	69.9	120	108	164	141	12.1	11.1	81.2	66.1
KwQ	47.3	46.9	109	**108**	14.6	12.4	85.8	70.3	44.2	42.2	103	106	11.7	11.1	80.7	65.6
KwQ_A	**31.7**	38.7	**94.5**	114	13.9	11.9	79.9	66.3	31.5	40.6	91.1	108	11.7	11.0	75.4	65.3
KwQ_{A+S}	32.9	38.8	94.4	119	14.0	12.0	80.2	66.2	33.4	42.2	90.2	110	11.4	10.9	75.1	65.7
DP	n.a.	n.a.	n.a.	n.a.	239	230	1498	1531	n.a.	n.a.	n.a.	n.a.	231	227	1483	1507
HDP	61.1	43.8	n.a.	n.a.	13.7	10.7	84.6	n.a.	57.4	39.4	n.a.	n.a.	11.0	9.7	76.9	66.1
HDP_A	65.3	47.1	n.a.	n.a.	12.3	9.7	81.8	67.3	64.3	46.1	n.a.	n.a.	10.9	9.7	79.1	66.5
VMK_B	62.1	62.6	183	160	12.4	11.2	155	146	48.9	47.7	164	137	**12.1**	10.9	151	141
$VMK_{B,shared}$	48.2	39.3	170	154	11.1	**8.7**	110	101	33.9	37.2	121	111	11.0	**8.7**	112	98.6
VMK	64.9	61.7	159	140	14.9	13.0	87.8	78.5	44.5	47.7	160	139	12.1	10.9	86.0	76.9
VMK_{shared}	44.3	42.6	150	140	**12.0**	8.7	66.8	56.3	35.1	37.8	117	99.8	11.0	8.7	66.2	55.5
$VMK_{shared,out}$	53.8	**45.3**	153	147	12.5	8.7	**66.9**	**56.1**	**32.1**	**33.6**	**115**	**118**	11.0	8.7	**66.5**	**54.9**
$VMK_{shared,dyn}$	60.5	54.4	234	223	14.6	10.8	96.4	88.9	51.8	54.6	190	191	12.2	10.6	95.1	86.7

Table D.6: Pipeline test with synthetic FMA load on a GTX TITAN

	4 Stages															
	Big								Small							
	Mixed				Packages				Mixed				Packages			
	et		vt		et		vt		et		vt		et		vt	
	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr
K	69.0	55.5	**83.2**	**70.7**	56.6	32.0	92.4	53.4	71.1	55.7	**85.6**	**69.7**	53.7	32.4	91.8	53.3
KwQ	76.6	65.3	199	161	56.6	32.0	92.6	53.4	77.9	70.5	205	160	53.6	32.3	92.3	53.1
KwQ_A	74.4	66.2	200	161	53.8	31.6	92.1	53.3	77.2	65.5	204	159	53.6	31.7	91.9	53.2
KwQ_{A+S}	72.2	63.7	201	158	**53.7**	31.7	92.2	53.2	76.7	66.2	200	159	**53.2**	31.8	91.9	53.0
VMK_B	60.7	55.8	239	201	56.6	31.0	99.4	54.0	56.1	51.0	233	199	56.2	30.7	98.4	54.2
$VMK_{B,shared}$	50.3	43.9	212	198	55.4	**29.3**	86.7	49.5	40.6	34.0	156	112	56.3	**29.6**	89.0	49.3
VMK	59.7	55.2	240	207	56.5	30.9	**86.7**	43.0	57.0	51.9	233	202	56.7	31.0	**86.7**	42.8
VMK_{shared}	**49.3**	**41.9**	208	177	56.5	30.8	87.4	42.3	40.8	33.2	151	126	55.9	30.6	88.0	**42.2**
$VMK_{shared,out}$	52.0	42.3	211	180	55.2	30.8	86.9	**42.1**	**38.9**	**31.1**	131	106	55.3	30.4	87.8	42.4
$VMK_{shared,dyn}$	75.8	69.8	319	295	57.3	33.3	88.1	42.4	72.7	56.5	236	216	59.9	32.1	89.4	44.8

Table D.7: Pipeline test with synthetic memory load on a GTX 580

	7 Stages															
	Big								Small							
	Mixed				Packages				Mixed				Packages			
	et		vt		et		vt		et		vt		et		vt	
	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr
K	101	72.0	**95.0**	**72.8**	135	88.3	423	332	100	72.7	**95.4**	**72.9**	118	91.5	425	331
KwQ	111	95.5	254	201	133	87.8	426	332	125	113	258	193	119	90.3	421	332
KwQ_A	110	107	247	196	119	87.0	421	329	126	109	254	192	120	86.9	422	331
KwQ_{A+S}	108	101	253	193	**120**	86.4	420	330	120	108	258	191	**119**	86.9	423	330
VMK_B	94.4	96.0	309	272	133	101	900	716	93.1	102	291	274	135	99.5	896	719
$VMK_{B,shared}$	97.1	90.8	291	253	127	**95.2**	911	702	93.3	73.1	130	201	126	**95.9**	878	688
VMK	90.7	94.9	275	244	136	99.6	**776**	723	95.4	87.7	267	251	134	98.8	**776**	721
VMK_{shared}	**94.6**	**84.3**	269	234	125	94.5	751	736	91.1	69.4	218	165	131	95.7	801	**707**
$VMK_{shared,out}$	101	93.1	275	240	125	93.1	773	**716**	**80.7**	**70.4**	224	184	131	95.0	803	700
$VMK_{shared,dyn}$	124	130	463	432	151	115	836	739	98.3	114	385	349	141	105	819	756

Table D.8: Pipeline test with synthetic memory load on a GTX 580

	4 Stages															
	Big								Small							
	Mixed				Packages				Mixed				Packages			
	et		vt		et		vt		et		vt		et		vt	
	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr
K	508	403	1218	1058	49.7	29.1	114	72.7	487	345	1157	836	55.0	29.3	114	72.0
KwQ	431	347	1015	932	50.2	31.7	114	72.4	408	331	960	809	54.6	31.4	114	71.6
KwQ_A	398	304	1008	835	51.6	31.3	113	72.9	395	300	969	691	53.6	31.0	114	72.8
KwQ_{A+S}	420	295	1036	858	49.2	31.5	113	72.9	399	319	959	678	53.5	31.1	114	72.8
VMK_B	367	354	961	916	62.3	39.8	169	125	405	331	966	879	62.9	40.3	169	124
$VMK_{B,shared}$	212	209	852	843	**38.1**	19.9	75.2	42.5	81.1	79.6	445	514	40.1	18.9	73.5	39.1
VMK	362	378	962	913	63.5	40.3	122	80.9	397	318	977	856	65.9	39.8	122	80.6
VMK_{shared}	225	209	**833**	825	44.5	19.2	**69.3**	32.1	81.7	84.0	459	505	**39.9**	18.9	66.7	**30.6**
$VMK_{shared,out}$	**195**	192	855	865	43.5	**19.1**	69.5	**32.0**	**49.4**	**53.1**	**249**	**243**	40.6	**18.7**	**66.4**	33.3
$VMK_{shared,dyn}$	198	**185**	908	**818**	46.0	21.4	80.2	44.2	259	313	503	457	46.3	21.5	82.3	47.5

Table D.9: Pipeline test with synthetic memory load on a GTX 680

	7 Stages															
	Big								Small							
	Mixed				Packages				Mixed				Packages			
	et		vt		et		vt		et		vt		et		vt	
	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr
K	1291	1022	1358	1111	102	94.8	881	863	1167	872	1597	1416	113	95.7	933	861
KwQ	900	908	1251	1192	109	95.1	523	474	831	695	1163	1038	112	93.6	523	473
KwQ_A	731	550	1232	1098	105	93.6	524	472	828	552	1169	869	106	93.7	519	476
KwQ_{A+S}	703	580	1300	1010	105	95.2	523	473	776	564	1113	850	106	92.9	524	474
VMK_B	544	550	1189	1073	152	139	1152	1074	682	588	1252	1169	151	123	1127	1068
$VMK_{B,shared}$	488	497	1195	1079	**80.4**	59.7	676	604	246	302	697	659	80.9	55.6	653	585
VMK	605	563	1133	1135	157	140	981	905	702	542	1141	1092	148	137	969	894
VMK_{shared}	480	492	**1083**	1119	82.1	66.5	**648**	566	248	287	698	692	**79.3**	56.4	624	**544**
$VMK_{shared,out}$	**473**	477	1083	1081	81.7	**61.8**	674	**601**	**275**	**302**	**786**	**712**	73.3	**54.6**	**617**	541
$VMK_{shared,dyn}$	436	**422**	1183	**1239**	107	71.7	832	778	374	413	1022	906	94.2	71.3	846	783

Table D.10: Pipeline test with synthetic memory load on a GTX 680

	4 Stages															
	Big								Small							
	Mixed				Packages				Mixed				Packages			
	et		vt		et		vt		et		vt		et		vt	
	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr
K	91.7	68.0	212	155	32.5	21.3	51.6	30.3	89.3	61.7	209	147	32.3	19.6	51.5	29.7
KwQ	31.7	**28.5**	112	94.1	32.6	21.2	51.5	30.3	30.9	30.5	109	103	32.4	19.6	51.6	29.7
KwQ_A	**28.5**	29.4	**93.9**	89.8	32.6	21.1	51.5	29.8	**26.6**	27.7	95.1	80.5	32.6	19.5	51.5	29.6
KwQ_{A+S}	28.9	29.1	94.0	**86.0**	32.7	21.1	51.4	29.9	27.3	27.9	94.2	81.7	32.6	19.4	51.5	29.5
DP	n.a.	n.a.	n.a.	n.a.	168	126	427	350	n.a.	n.a.	n.a.	n.a.	170	123	429	348
HDP	59.5	39.7	n.a.	179	43.4	19.8	69.3	30.9	56.8	35.0	n.a.	180	46.9	19.1	67.8	30.3
HDP_A	65.0	45.0	n.a.	239	45.5	19.1	68.2	30.4	65.1	43.2	n.a.	235	46.1	19.2	67.8	30.3
VMK_B	48.7	44.3	151	144	**31.3**	17.7	64.4	35.8	34.7	33.0	155	137	**31.4**	17.9	63.9	35.6
$VMK_{B,shared}$	37.4	31.4	115	126	31.9	17.4	52.3	27.6	30.3	24.9	88.4	92.6	31.8	17.5	52.3	27.8
VMK	47.9	43.6	157	145	31.4	20.9	50.5	27.2	38.7	34.2	111	141	31.5	17.8	**50.2**	27.0
VMK_{shared}	36.4	31.8	117	131	32.0	17.6	50.9	**24.1**	29.8	25.6	90.5	93.3	31.6	17.5	50.2	**24.0**
$VMK_{shared,out}$	35.9	31.1	116	131	31.8	17.7	51.0	24.2	26.8	**21.6**	**77.5**	**75.4**	31.7	17.6	50.8	24.2
$VMK_{shared,dyn}$	48.5	43.8	142	176	31.6	**17.4**	**50.5**	26.3	48.1	47.0	112	119	36.3	**17.5**	50.7	29.6

Table D.11: Pipeline test with synthetic memory load on a GTX TITAN

	7 Stages															
	Big								Small							
	Mixed				Packages				Mixed				Packages			
	et		vt		et		vt		et		vt		et		vt	
	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr	er	vr
K	215	147	293	168	77.6	66.0	506	444	212	139	290	163	75.9	65.3	505	449
KwQ	55.4	**58.1**	130	103	77.4	66.1	506	449	49.9	56.8	126	109	77.2	66.0	505	448
KwQ_A	**36.9**	44.5	**108**	94.1	77.8	66.3	506	443	**40.3**	49.3	106	89.4	77.3	64.9	506	441
KwQ_{A+S}	37.4	43.3	108	**95.3**	77.3	66.2	504	441	41.9	47.7	107	89.2	78.0	66.1	507	440
DP	n.a.	n.a.	n.a.	n.a.	361	329	2311	2149	n.a.	n.a.	n.a.	n.a.	354	323	2293	2097
HDP	97.5	80.7	n.a.	n.a.	97.1	77.2	n.a.	n.a.	98.1	63.8	n.a.	n.a.	104	74.4	n.a.	n.a.
HDP_A	107	73.1	n.a.	n.a.	101	79.3	n.a.	n.a.	111	71.7	n.a.	n.a.	103	74.1	n.a.	n.a.
VMK_B	74.2	66.9	149	184	**75.2**	62.0	534	478	58.8	57.6	150	186	**75.1**	61.5	530	476
$VMK_{B,shared}$	57.6	59.3	157	182	76.1	60.8	508	444	52.4	45.3	118	147	75.5	61.4	507	445
VMK	77.0	67.2	183	167	75.2	61.4	500	448	60.2	52.4	133	159	75.6	61.9	**493**	449
VMK_{shared}	58.9	52.6	144	156	75.9	62.0	517	**469**	52.8	45.3	114	122	75.7	61.6	528	**479**
$VMK_{shared,out}$	59.3	52.9	146	171	76.8	61.8	517	467	48.7	**37.9**	**116**	**130**	76.8	60.9	527	479
$VMK_{shared,dyn}$	83.4	58.4	183	242	77.5	**61.6**	**501**	451	66.0	55.6	157	188	76.0	**62.4**	502	449

Table D.12: Pipeline test with synthetic memory load on a GTX TITAN