Dipl.-Ing. Bernhard Kerbl, BSc

Load Balancing for Hardware and Software Rendering on the Graphics Processing Unit

Dissertation

to achieve the university degree of

Dr. techn.

Doctoral Program: Computer Sciences

submitted to

Graz University of Technology

Supervisors

Prof. Dr. Dieter Schmalstieg and Ass.Prof. Dr. Markus Steinberger

Evaluator

Prof. Dr. Michael Doggett

Institute of Computer Graphics and Vision
Head: Prof. Dr. Dieter Schmalstieg

Graz, October 2018

"Speed is key."

Seán William McLoughlin

Affidavit

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 indicated all material which has been quoted either literally or by content from the sources used.

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 entnommenen Stellen als solche kenntlich gemacht habe.

Ort	Datum	Unterschrift

Abstract

The key to coping with the massive computational demands of rendering large 3D scenes in real-time lies in parallelization. To provide the necessary computing power, a modern graphics processing unit (GPU) features thousands of cores. However, channeling this power into competitive performance demands effective load balancing strategies to make sure all cores are best utilized at all times. Static load balancing distributes the workload according to a fixed, predefined scheme based on the properties of each individual work package. In contrast, dynamic load balancing can take additional system parameters of the GPU into account, *e.g.*, the current fill rate on each individual processor. Doing so increases the scope of possible load balancing methods but also leads to higher conceptual complexity and communication overhead.

This thesis analyzes and extends practical use cases of both dynamic and static load balancing strategies fit for execution on the GPU architecture. We examine this topic by considering the GPU in its two key roles for modern computer science: first, as a ubiquitous, general-purpose co-processor for massively parallel software applications; second, as a powerful, highly efficient sort-middle rasterization pipeline for graphics content. Focusing on the example of software rendering applications, which are commonly bound by particularly tight runtime constraints, we aim to enable custom scheduling, task aggregation and even prioritization for a broad range of time-critical procedures. Furthermore, we analyze non-trivial load balancing behavior and policies as they are being used for processing data in the conventional hardware rendering pipeline.

The obtained results prove the significance of sophisticated scheduling and load balancing for the performance of computer graphics applications. Based on appropriate guidelines for the design of algorithms and data structures, tailored towards the peculiarities of today's GPUs, we apply novel insights to optimize work distribution for parallel software rendering and demonstrate adaptive behavior to make optimal use of available computing power. We also show how in-depth knowledge about internal load balancing enables optimization of input to minimize shading effort during hardware rendering.

Kurzfassung

Der Schlüssel zur Bewältigung des wachsenden Bedarfs an Rechenleistung bei der grafischen Darstellung von komplexen, dreidimensionalen Szenen liegt in der Parallelisierung. Um die nötige Leistung zu erbringen, verwenden moderne Grafikprozessoren (GPUs) mehrere Tausend Rechenkerne gleichzeitig. Die optimale Nutzung dieses Potenzials erfordert jedoch eine gerechte Verteilung der gesamten Arbeitslast. Statische Ansätze für den Lastausgleich folgen einem fixierten, vordefinierten Schema, basierend nur auf den Eigenschaften der einzeln anfallenden Arbeitspakete. Beim dynamischen Lastausgleich können zusätzliche Systemparameter (*e.g.*, der momentane Status einzelner Rechenkerne) im Verteilungsprozess berücksichtigt werden. Dynamische Methoden erweitern daher den Gestaltungsraum möglicher Strategien, erfordern jedoch zusätzliche Mess- und Kommunikationsprozesse.

In dieser Dissertation wird die praktische Anwendbarkeit von dynamischen und statischen Methoden zum Lastausgleich auf der GPU analysiert und erweitert. Die Behandlung dieses Themas erfordert die Betrachtung der GPU in ihren wesentlichen Rollen für die wissenschaftliche Forschung: erstens, als leistungsstarker, hocheffizienter und zunehmend allgegenwärtiger Co-Prozessor für parallele Berechnungen; zweitens, als hochoptimierte Komponente für die Generierung von Rasterbildern aus grafischen Inhalten.

Ziel ist die Ermöglichung maßgeschneiderter dynamischer Strategien für den Lastausgleich bei komplexen, üblicherweise zeitkritischen Abläufen, wie etwa Software Rendering. Außerdem untersuchen wir nicht-triviale, statische Methoden zum Lastausgleich, wie sie bei konventionellem GPU Rendering mittels Rasterisierung zum Einsatz kommen. Die so gewonnenen Einsichten und Resultate zeigen die Bedeutung guter Verteilungsstrategien für die Leistungsfähigkeit moderner Grafikapplikationen auf. Neue Einblicke ermöglichen die Umsetzung von verbesserter Lastverteilung und infolgedessen adaptives Verhalten auf der GPU, basierend auf dafür geeigneten Richtlinien. Des Weiteren kann sogar die Leistung in konventionellen GPU Renderprozessen durch ein klares Verständnis interner Ausgleichsstrategien und entsprechender Datenaufbereitung nachweislich verbessert werden.

Acknowledgements

I would like to express my gratitude for the care, attention and effort invested in me over the last few years by superiors and comrades alike. I want to thank my supervisor, Prof. Dieter Schmalstieg, whose resourcefulness, ingenuity and creativity are lauded by those who have had the pleasure of working with him. His vast expertise, paired with a readiness to consider novel fields of research, routinely rewards students who pick up on his ideas with academic decorations. Dieter's sage qualities have been perfectly complemented by my co-supervisor, Ass. Prof. Markus Steinberger, whose seemingly limitless vigor is only exceeded by his wit and skillfulness. Markus' quest for ever greater achievements unites and propels those who are willing to devote their time and effort to the conception of new landmarks in computer science.

Special thanks go to my colleagues who let me participate in their research and contributed to mine. I want to specifically thank Denis Kalkofen, Michael Kenzel, Peter Mohr, Jörg Müller, Markus Tatzgern, Michael Donoser, Philip Voglreiter, Wolfgang Tatzgern, Elena Ivanchenko and Bernhard Kainz. I would also like to thank Lorenz Jäger, Patrick Kasper, Alexander Isop, Okan Erat, Laurenz Theuerkauf, Daniel Brajko, Alexander Skiba, Andreas Wurm, Christian Lesjak, Christoph Wörgötter, Dietmar Maurer, Martin Oswald, Johannes Iber, Pedro Boechat and Franz Leberl, who gave me the motivation, help and support needed to fulfill my academic and educational duties as a researcher and university teaching assistant.

Furthermore, I want to thank Michael Doggett for agreeing to review this thesis, as well as for his guidance during my research with the Lund University Graphics Group.

I am grateful for having been granted two exceptionally caring parents, Ulrike and Reinhold, who never stopped supporting and encouraging me on my way. With honesty and prudence, they have contributed to the progress of my studies in more ways than I can recount.

This work was supported by the German Research Foundation (DFG) grant STE 2565/1-1 and the Austrian Science Fund (FWF) grant I 3007.

Contents

List of Figures

1. Introduction

Contents

As modern society embraces the arrival of the digital age, the amount of data we produce, consume and rely on every day keeps growing in quantity and complexity, with no signs of slowing down. Our daily routines are increasingly dependent on powerful tools and procedures to handle, distribute and convey new information as quickly as possible. While advances in semiconductor fabrication continue to follow Moore's Law, increases in clock speed have become stagnant due to physical limitations. Instead, hardware and software have turned towards parallelization as an answer to the ever-growing demand for more computing power. The graphics processing unit (GPU) plays an integral role in this scenario, as it has progressively evolved into a massively parallel hardware architecture that consumers and researchers alike are aiming to harness for performing diverse, computation-intensive routines.

1.1. Evolution of the Graphics Processing Unit

The history of the GPU is one of continuous transformation. As the requirements and objectives of its users and programmers changed, so did the GPU in an effort to satisfy their demands. From its early days of bit blitting to its modern-day role as an exceedingly fine-tuned rendering powerhouse, more than three decades have passed. During this time, the transition from two-

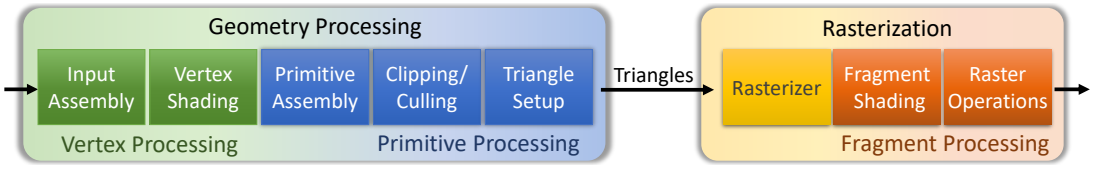


Figure 1.1.: Simplified DirectX 9-style rendering pipeline

to three-dimensional content marks a major shift in focus for the graphics community and the rendering pipeline. The ability to handle polyhedral geometry relieved visual content creators from the effort of providing renderings for their artwork from all feasible viewing perspectives. Instead, the transformation of vertices outlining a 3D object, according to a spectator's particular view, could now be performed by a dedicated geometry processing stage. Subsequent stages of the graphics pipeline would then perform the assembly and rasterization of the resulting 3D primitives (see Figure 1.1).

The key to handling increasingly complex 3D scenes in real-time lies in parallelization. At runtime, the GPU continuously fetches adequate work packages from a data stream or queue and distributes them to available processors, thereby achieving trivial, yet effective load balancing. To increase performance, manufacturers have continuously raised the number of built-in processors (or *shading units*) with every new generation of GPU models, and continue to do so to this day. Although vertex transformation was initially handled by the central processing unit (CPU), the performance benefits of independently processing vertices in parallel soon provided sufficient reason to expand the capabilities of the GPU accordingly. Furthermore, memory transfer for camera updates on static geometry was greatly reduced, as vertex data could reside on the GPU and simply be transformed anew with an appropriate matrix. Hence, early GPU shading units were grouped into separate physical modules and featured a minimal, specialized instruction set to fulfill their respective task: vertex transformation or fragment shading.

Little by little, the desire to make rendering more convenient and less restrictive led to the insertion of additional access points to the pipeline, enabling elevated configurability and, eventually, programmability. Most notably, the responsibility for computing vertex and pixel information was transferred from hardware to custom shader programs meant to be executed directly on the GPU: Shading languages such as HLSL and GLSL emerged for the sole purpose of writing code to compute the final state of individual vertices or fragments from incoming vertex attributes and auxiliary parameters.

In the following hardware generations, the rendering pipeline was extended even further: adding geometry and tessellation shaders made the visualization of highly detailed models more convenient and memory-efficient. These additions were facilitated by the advent of the unified shader architecture: instead of requiring different hardware modules to take care of individual stages, shading units were made more versatile and capable of processing any shader, regardless of where in the pipeline it occurs. The augmented capabilities and instruction sets of these modern shader units paved the way for the utilization of GPUs as ubiquitous, general-purpose co-processors.

1.2. The GPU as a General-Purpose Co-Processor

The documented path of GPU applications from the early prototypes to contemporary models manifests many smaller branches, indicating experimental features that were tried, tested and usually discontinued. However, one particular fork in this road remains to this day: In addition to hardware rendering, the GPU has attained a significant alternative purpose as a massively parallel, general-purpose co-processor for handling large, homogeneous workloads, typically overseen and controlled by the CPU. Early on, carefully formatting the input feed for vertex and fragment shaders provided users with ways to run computations in parallel on the GPU that were unrelated to rendering.

These efforts were soon answered by the introduction of compute shaders to rendering APIs and even specialized compute APIs to write parallel procedures for execution on the GPU. Two popular examples are given by the vendor-specific Compute Unified Device Architecture (CUDA) for Nvidia models and OpenCL for AMD. With these tools, it has become relatively easy to formulate complex algorithms as functions that are concurrently called by thousands of light-weight GPU threads under a single-instruction-multiple-data (SIMD) directive. Code for parallel execution can be written with C/C++ syntax, and many fundamental instructions (*e.g.*, atomic integer operations) are either built-in or can be implemented based on available primitives. Compute APIs and shaders operate in a separate context and expose far more direct control over execution configuration, synchronization and communication. For threads running in lockstep on the same compute unit, they usually provide optimized methods for exchanging or communicating results efficiently. On the other hand, they are barred from access to specialized hardware features that play a crucial role in high-performance rendering, such as the render output unit (ROP) or control over collaborating rasterization clusters.

Developers who use compute shaders or APIs are responsible for configuring the launch of a job such that a sufficient number of calls to the parallel procedure will be scheduled. The scheduling of instructions on the GPU is dictated by a simple, hardwired first-in-first-out (FIFO) scheduler, which cannot be influenced dynamically. Such a regime only permits very coarse-granular control of execution on the device. Using a contemporary rendering API such as OpenGL or Direct3D, commands are simply streamed to the GPU and executed more or less in order. Similarly, compute jobs are dispatched and inserted into opaque queues, from which they are then launched onto the device. Once a job has been dispatched, it cannot be modified and must run to completion. Consequently, the GPU is rarely considered as a viable option for complex compute applications that depend on sophisticated scheduling for, *e.g.*, enabling fair resource sharing, exhibiting adaptive behavior, or adhering to a strict time plan. This lack of fine-granular control has led researchers to look for ways to work around the rigid hardware scheduling policies.

1.3. Non-trivial Load Balancing on the GPU

By denying access to low-level hardware scheduling decisions for compute jobs, manufacturers impede developers' abilities to implement sophisticated scheduling strategies. Prioritization, task aggregation and quota-driven execution are only some examples for the concepts that are unavailable with conventional compute job designs. Using the example of software rendering—which is often bound by real-time constraints—it has been shown that prioritization and adaptive scheduling of workloads can be very beneficial to the performance of graphics applications (Hachisuka et al., 2008; Overbeck, Donner and Ramamoorthi, 2009; Rousselle, Knaus and Zwicker, 2011). Several resourceful members of the computer graphics community have previously suggested ways to circumvent built-in scheduling and instead introduce custom work distribution schemes. Unfortunately, the presented approaches are either highly specialized, lack options for fine-granular prioritization or incur an immense overhead, effectively nullifying any potential performance gain. However, it stands to reason that a topic as recent and powerful as GPU programming warrants additional effort to pursue approaches for fine-granular, dynamic scheduling and thereby advance the status quo of massively parallel computing. Perhaps finding a solution to these issues becomes easier as we contemplate examples of non-trivial load balancing found in the GPU's hardware rendering pipeline and analyze how those are handled internally.

Based on the observation that vertices are usually referenced several times throughout a mesh, GPU vertex processing strives to reduce the number of redundant shader invocations through vertex reuse. Historically, this feat was achieved by the post-transform vertex cache. However, the dependence on a centralized caching structure is at odds with the massively parallel design of modern GPU hardware. Furthermore, suggesting that a reusable vertex can only be detected after a thread has been tasked with shading it implies wasted cycles. Even worse, entire warps would be forced to execute shading in lockstep if even a single assigned vertex is unavailable for reuse. A much more intuitive solution would be to assume support for basic load balancing strategies in the hardware rendering pipeline. This way, incoming vertex streams could be analyzed and appropriately split to produce work packages with an adequate workload for utilizing warps under reuse. Whether these capabilities are actually present in hardware and how vertex reuse is managed internally by the GPU remains to be seen.

A similar situation arises with regard to the transition of data from geometry processing to the rasterization stage. The hardware rendering pipeline is, with global consent, considered to be a *sort-middle* architecture: assembled 3D primitives are sorted into image-space bins before rasterization occurs, thus permitting each rasterizer to exclusively modify a portion of the frame buffer without the need for costly synchronization. While this seems perfectly reasonable considering the cost of global memory access and communication overhead with thousands of concurrent threads, it raises the question of how exactly this intermediate sorting step is implemented. To avoid bottlenecks from a small number of overburdened processors, assignment of primitives must employ a reliable load balancing strategy to uniformly distribute the workload among all active rasterizers.

1.4. Research Objectives

It is with these deliberations in mind that we set out to take a closer look at custom, high-level scheduling strategies for the GPU in compute mode and to evaluate whether they can be implemented in software applications that depend on prioritization, without sacrificing critical performance. Furthermore, we aim to devise experimental strategies for identifying and exploiting the mechanics of to-date undisclosed load balancing policies incorporated into the hardware rendering pipeline. We define the following research objectives:

- Summarize and learn from the achievements of previous methods for enabling custom scheduling on the GPU. Our focus lies with approaches that enable work prioritization, ideally as fine-granular as possible.

- Analyze the most essential and common modules incorporated into these software scheduling systems and assess their potential for optimization. As we improve the performance of the basic building blocks, we increase the chance of high performance in the final assembled system.

- Suggest a load balancing solution that enables efficient work prioritization and provides a high level of support for arbitrary scheduling policies. Furthermore, we strive to minimize overhead in order to ensure and demonstrate applicability even for compute jobs that are usually bound by real-time constraints, *e.g.*, software rendering on the GPU.

- Identify mechanisms used internally by the modern GPU to achieve adequate load balancing at various stages of the hardware rendering pipeline. The obtained knowledge should advance our understanding of how to optimally utilize the available hardware modules and fill in some of the blank spots in our conceptual map of the pipeline layout.

- Utilize the novel information on internal GPU load balancing to improve the performance of classic sort-middle rendering. Ideally, this should apply to both the hardware pipeline, as well as software implementations: For the first, we hope to find new ways of optimizing input data to exploit our new-found insights. For the latter, we expect that hardware solutions for non-trivial load balancing can serve as a guideline for implementing effective methods in GPU software rendering as well.

Our approach to each of these topics, experimental results and the insights we have obtained during our research were, to the best of our abilities, formulated and documented in the upcoming chapters of this thesis.

2. Related Work

Contents

A massively parallel device such as the modern GPU provides several opportunities for applying load balancing and optimization schemes to ensure that available cores are reasonably utilized at runtime. In the context of parallel program execution, high processor utilization is generally preferable, as it either minimizes the runtime required for processing a fixed workload or, inversely, maximizes the amount of work that can be done within a given time budget. Previous work has already demonstrated the effectiveness and ample usefulness of fully dynamic as well as static load balancing methods for GPU applications.

Terminology In the following, we use the nomenclature conventions established by the Nvidia/CUDA execution model to refer to hardware modules and routines: Computation on the GPU is based on SIMD execution in small groups of threads called *warps* that operate in lockstep. A compute job is described by a *kernel* that defines a grid of *blocks*, with each block consisting of the same number of warps, that all fit on one of the GPU's *streaming multiprocessors* (SM). For topics relating to the rendering pipeline, we refer to stages and shaders according to their names in standard OpenGL documentation.

2.1. Dynamic Load Balancing & Prioritization

Focusing on the GPU as an increasingly programmable general purpose co-processor for parallel software applications, previous work has shown that complex procedures can be divided into work packages, which may then, in turn, be distributed and managed on the device to manifest custom dynamic scheduling behavior. These approaches enable high-level techniques—conventionally reserved for CPU-side execution—to be implemented on the GPU, such as task prioritization, adaptive processing and dynamic load balancing. The foundation for these techniques is usually provided by an underlying parallel data structure that efficiently stores, handles and promotes available work packages, the most common choice being a queue. Hence, in the interest of leveraging high-level dynamic load balancing techniques on the GPU, it is essential to advance our understanding of effective queuing strategies for these massively parallel devices. Naturally, the benefits of doing so also affect the applicability of load balancing to software rendering applications. Given a limited time frame, priority-based adaptive rendering routines boast a higher image fidelity than naïve alternatives. This is commonly achieved by directing more computing power to any domain (*e.g.*, image region) that requires more thorough processing than others, *i.e.*, selective prioritization.

2.1.1. Work Distribution Schemes on the GPU

For current GPU architectures, a variety of external priority schedulers exist. They can be viewed as an extension to the driver, controlling which kernels are forwarded to the GPU at which point in time. Timegraph (Kato, Karthik Lakshmanan et al., 2011), for example, delays and reorders kernels that are being sent to the GPU according to deadlines. More intricate CPU-side scheduling systems explored priority queues (Elliott and Anderson, 2012) or directed acyclic graphs (Membarth et al., 2012) for managing kernels, as well as the possibility of scheduling across multiple CPU and GPU instances (Wen, Z. Wang and O'Boyle, 2014). The major limitation of these approaches is that they consider kernels as immutable, indivisible entities: Memory transfers (Kato, K. Lakshmanan et al., 2011) and kernels (Basaran and Kang, 2012) can be split into smaller jobs to reduce the time until the GPU is responsive again. With fine-granular scheduling of blocks instead of entire kernels, utilizing knowledge about how many multiprocessors are available, a better real-time scheduling performance can be achieved (H. Lee and Al Faruque, 2014).

Yet, all these approaches influence GPU scheduling only indirectly, which leads to several disadvantages: a long delay between making a scheduling decision on the CPU and its effect on the GPU, the need to frequently synchronize CPU and GPU, and the inherent inefficiency of submitting kernels too small to fully occupy the device. Additionally, dynamic priorities and work generation on the GPU are not supported by these approaches. Recent GPU architectures can execute multiple kernels concurrently (Nvidia, 2012) given that sufficient resources are available. This feature relies on multiple work queues situated on the GPU, each feeding all available streaming multiprocessors. While priorities are not supported for these queues, future architectures might add them, as outlined in the provisional OpenCL 2.1 standard (Khronos-Group, 2015). However, these priorities are assumed to be static and only applicable to entire kernels.

The *persistent threads* strategy (Aila and Laine, 2009) aims to cope with this lack of fine-grained task scheduling mechanisms in the kernel-based GPU execution model. Their concept employs a software queuing structure and persistent threads that fill up the entire GPU and run in an infinite loop. In every iteration, each thread consumes available work items from the queue, until no more items remain. Cederman and Tsigas (2008) were the first to build a load balancing system based on this approach. Persistent threads in combination with work queues have been used in a variety of other application areas: sparse matrix-vector multiplication (Bell and Garland, 2009), sorting algorithms (Satish, Harris and Garland, 2009), scan algorithms (Breitbart, 2011), and construction of kd-trees (Vinkler et al., 2015). Although a single queue is sufficient to achieve simple work distribution, multiple queues are often used for advanced scheduling mechanisms, such as inserting tasks from the CPU (L. Chen et al., 2010), task donation (Tzeng, Patney and Owens, 2010), stealing (Chatterjee et al., 2011), or managing different task types in a single *megakernel* (Steinberger, Kenzel, Boechat et al., 2014).

To add multi-tasking to a persistent threads approach, all tasks have to be compiled into one large megakernel (Hargreaves, 2005). In their simplest form, megakernels are written as one large switch clause, with threads deciding at each iteration which case (*i.e.*, task) to execute. While there are certain downsides to megakernels (Laine, Karras and Aila, 2013), they are often outweighed by better scheduling and the potential to exploit data locality (Steinberger, Kenzel, Boechat et al., 2014). Megakernels have been used in the Optix raytracing framework (Parker et al., 2010) as well as the dynamic task scheduling frameworks of Softshell (Steinberger, Kainz et al., 2012) and Whippletree (Steinberger, Kenzel, Boechat et al., 2014).

While not their main focus, the latter two also provide ways of prioritizing workloads on the GPU: Softshell uses a monolithic queue, which can be progressively sorted, slowly moving high priority tasks to the front of the queue. Whippletree uses multiple queues to concurrently schedule warp and block level tasks, and can explicitly collect tasks of the same type (task aggregation). Whippletree further allows for a prioritization of individual queues over others, which can be used to, *e.g.*, keep their lengths manageable at runtime.

2.1.2. Concurrent Queue Designs

With queues playing such an integral role in the implementation of work distribution software for the GPU, it seems imperative that we identify effective, concurrent queuing algorithms that minimize overhead and thus maximize scheduling benefit. Although alternative structures other than queues have been successfully employed for simpler work distribution schemes (Arora, Blumofe and Plaxton, 1998; Hendler, Lev et al., 2006), doing so usually implies the abandonment of linearizable first-in-first-out (FIFO) ordering.

FIFO Queues and Linearizability Arguably, the FIFO queue design, which defines a *head* from which items are drawn and a *tail* for appending items, is the most common choice. FIFO naturally lends itself to work distribution, because it implicitly maintains tasks in the order in which they were submitted. In case of a single producer and single consumer scenario, a strict FIFO ordering, as described in the original algorithm by Lamport (1983), *FastForward* (Giacomoni, Moseley and Vachharajani, 2008), or *MCRingBuffer* (P. P. C. Lee, Bu and Chandranmenon, 2009), is easy to achieve.

In a massively parallel scenario, however, FIFO in its original sense is no longer applicable, since many threads can concurrently interact with the queue. This fact gives rise to the concept of linearizability, which can be used to prove observable FIFO behavior when operations overlap temporally. In short, linearizability can be understood as the constraint that an external observer, observing only the abstract data structure operations, gets the illusion that each of these operations takes effect instantaneously at some point between its invocation and its response (M. P. Herlihy and Wing, 1990). Although it is not necessarily required for achieving effective work distribution, ensuring linearizable FIFO behavior has become a quality seal and defines the gold standard for sophisticated concurrent queue designs.

Lock-Freedom In a multi-threaded environment, a *blocking* algorithm involves procedures that can arrive at a system-wide impasse if one or more threads cannot progress due to failure or suspension. In contrast, *non-blocking* algorithms are guaranteed to avoid such gridlocks by ensuring either *lock-freedom* or *wait-freedom*: the first constitutes that at least one thread can proceed at all times and thereby ensure system-wide progress, while the latter guarantees continual progress for every single participating thread. The available literature on concurrent queues has a strong focus on lock-freedom, which is often held as key to performance in concurrent systems.

Previous Concurrent Queuing Algorithms One way of constructing a concurrent queue design is given by using a *linked list*. Valois (1994) provided one of the first lock-free, link-based queues using *compare-and-swap* instructions. Problems with the original design were later corrected by Michael and Scott (1995), although the corrections greatly diminish its practical value. An alternative is provided by the authors in the Michael-Scott queue (Michael and Scott, 1996), which is still among the most popular lock-free concurrent queues. The same authors further present a blocking queue that supports concurrent insertion and removal using two locks. The baskets queue by Hoffman, Shalev and Shavit (2007) presents a variation on the Michael-Scott queue, exploiting the fact that no binding order can be defined for elements that are concurrently inserted, thus any ordering is equally valid.

Array-based queues employ a continuous array of elements, commonly operated as a ring buffer. The early work by Gottlieb, Lubachevsky and Rudolph (1983) introduced the first array-based queue that scales linearly to a large number of cores thanks to a fine-grained locking approach. In addition to head and tail pointers, it uses two counters to track concurrent enqueues and dequeues. However, due to its rather simple design, their queue is not linearizable (Blelloch et al., 2003). Orozco et al. (2012) addressed these issues by presenting the circular buffer queue, which avoids the additional counters, but acts fully blocking during enqueue and dequeue. As an alternative, they propose the high throughput queue, which returns to the problematic two-counter approach of Gottlieb, Lubachevsky and Rudolph. Valois (1994) proposed a lock-free, array-based queue that relies on compare-and-swap on non-aligned memory addresses, which is not supported by common processor architectures, thus rendering it impractical. The ring buffer by Shann, Huang and C. Chen (2000) appears to be the first practical lock-free, array-based queue. Also lock-free, Tsigas and Zhang (2001) presented their queuing

solution, which reduces contention by updating parts of the queue only periodically. Recently, Morrison and Afek (2013) have proposed a lock-free linked concurrent ring queue, which consists of multiple smaller arrays and avoids contention by using fetch-and-add over compare-and-swap. Yang and Mellor-Crummey (2016) presented a similarly segmented, wait-free queue that uses a fast-path/slow-path dynamic to avoid stalling threads.

Contrary to the prevailing trend of lock-freedom, Scogland and Feng (2015) have presented a blocking array-queue, built on top of a ring buffer ticket system, that can easily be ported to the GPU. While their approach offers high performance, it blocks execution when the queue runs full or empty, and only a single thread in each warp is allowed to interface with the queue. An alternative, non-blocking interface has been proposed to circumvent this, but its implementation entails a drastically reduced queuing performance.

Apart from enabling general work scheduling strategies, priority queues can be particularly helpful to computer graphics in the context of adaptive rendering procedures. In these particular applications, one or more queues are used to organize and continuously promote the execution of tasks or jobs that are currently expected to have the best effect on image quality.

2.1.3. Adaptive and Prioritized Rendering

Adaptive rendering follows the idea of smartly managing limited computational resources, so as to achieve high image fidelity with reduced computational effort. A prominent example of applying this principle is given by the REYES architecture (Cook, Carpenter and Edwin Catmull, 1987): Large input patches are progressively subdivided (or *split*) into multiple, smaller patches, according to guiding mathematical principles for computing reducible, continuous surface structures (*e.g.*, Bezier splines). Patches found to be wholly occluded or invisible can be immediately culled. Eventually, patches are *diced* into primitives. If a visible primitive is deemed fine enough to satisfy visual demands (*e.g.*, its screen-projected bounds fall below the Nyquist rate), it is shaded and its color value is added to the output image. This procedure is inherently adaptive since the number of splits per patch is not fixed, but changes depending on their extent and proximity to the camera. Furthermore, for adhering to a fixed runtime budget, a priority queue can be used to ensure that big patches are subdivided first, so that at any given time during the procedure, all projected patches are more or less equally fine.

In the first full REYES rendering pipeline for the GPU, *RenderAnts*, Zhou et al. (2009) employ a parallel method for subdivision, which was first described by Patney and Owens (2008). Their approach repeatedly launches CUDA kernels to split and bound input patches, deciding at the end of each iteration, which are to be culled, split or shaded in the next. To accommodate the growing number of items to process as patches are split repeatedly, the authors maintain a queue to store geometric details and assigned task for each patch. Patney, Ebeida and Owens (2009) further presented similar solutions for Catmull-Clark subdivision (E. Catmull and Clark, 1998) on the GPU. Steinberger, Kenzel, Boechat et al. (2014) demonstrate an exemplary REYES-style micropolygon rendering approach that avoids multiple kernel launches by using a persistent megakernel. Their approach was later extended by Sattlecker and Steinberger (2015) to produce advanced visual effects, including displacement mapping, motion blur and depth-of-field.

Another notable example of adaptive rendering is given by the concept of "a posteriori" Monte Carlo sampling techniques for path tracing (Zwicker et al., 2015). While micropolygon rendering targets geometry subdivision, adaptive path tracing operates on the image domain instead. Pioneered by early attempts to reduce sampling rates while preserving image quality (Mitchell, 1987; Ward, Rubinstein and Clear, 1988; Guo, 1998), these methods aim to reduce notorious high-frequency noise by dynamically prioritizing image segments with high estimated error: For each segment of the output image, new samples are created depending on current variance, in order to achieve uniform fidelity. Hachisuka et al. (2008) abstract the concept to the k-dimensional space, where each dimension corresponds to an independent variable that affects the image function (*e.g.*, time, depth-of-field distortion). Error values are then computed and updated for all k-dimensional segments contained by a kd-tree data structure. Trying to avoid the "curse of dimensionality", Overbeck, Donner and Ramamoorthi (2009) instead employ a 2D wavelet basis to obtain image error estimates and perform a smooth reconstruction. Rousselle, Knaus and Zwicker (2011) and Li, Y.-T. Wu and Chuang (2012) expand on this concept by supporting not only wavelets but rather providing a variety of different filters. For each pixel and iteration, filters are then chosen for their ability to minimize the error. Moon, Carr and S.-E. Yoon (2014) successfully apply local regression and use a reduced subspace of features to guide adaptive sampling. All of the above approaches are based on priority queues, which they use for storing image segments and updating the ordering within depending on current sampling error estimates. By continuously dispatching new samples to the segment that is currently the head of the queue, regions

with high divergence receive more samples than others. Doing so improves convergence rates significantly and reduces the time required to produce high-fidelity renderings, compared to naïve uniform sampling. The dependence of these adaptive algorithms on priority queuing, however, renders an effective implementation in a massively parallel environment challenging.

In an effort to attain similar benefits for path tracing on the GPU, Liu, J.-Z. Wu and Zheng (2012) and Liu and Zheng (2013) proposed parallelizable algorithms with adaptive behavior. Their solution first performs analysis on a low number of initial samples to determine a one-time, static allocation of secondary samples per image region. Based on this initial estimate, image regions are then rendered independently by individual blocks, closely mimicking the approaches by Hachisuka et al. (2008) and Overbeck, Donner and Ramamoorthi (2009). Other, more sophisticated 'a priori' methods (Zwicker et al., 2015; Lehtinen et al., 2012; Mehta et al., 2014; Kettunen et al., 2015) can be similarly adapted and therefore lend themselves to execution on the GPU. A priori load balancing solutions for improved image quality are straightforward to implement, but unable to dynamically react to sudden changes in the error estimate, which reflects the downside of static load balancing.

2.2. Static Load Balancing & Hardware Rendering

In contrast to dynamic scheduling techniques, more specialized, static methods for balancing GPU workload are an integral part of hardware rendering. Given the classic example of a sort-middle pipeline, we can select individual stages and examine the current conventions and potential improvements for optimizing load at runtime. In this thesis, we consider two specific pipeline tasks: vertex processing and primitive rasterization. For vertex processing, it is historically assumed that a centralized post-transform vertex cache enables reuse of previously shaded vertex information. Several methods have been proposed that exploit this assumed GPU hardware feature to minimize the number of vertex shader invocations. However, with the inexorably increasing parallelism found in graphics hardware, it seems illogical that modern devices would still rely on such a highly contended resource.

By identifying more reasonable, inherently parallel approaches to vertex reuse, we gain new insights into how static load balancing for rendering occurs in hardware today. Additionally, doing so provides us with a starting point for implementing effective vertex processing in software rendering applications for GPUs (Kenzel, Kerbl, Tatzgern et al., 2018).

Regarding primitive rasterization, we claim to provide the first comprehensive analysis of how spatial layouts of screen-space bins can influence the expected performance when rendering realistic input data in a sort-middle pipeline.

2.2.1. Vertex Processing

The average vertex is referenced six times in a well-connected triangle mesh. Both hardware and software solutions have been presented over time in an effort to save on redundant computations through reuse of already transformed vertices, with the goal of significantly reducing vertex processing load and overall rendering runtime. Early attempts approached the issue as a problem of compressing and decompressing the input geometry, which led to the conception of triangle strips as well as algorithms that turn arbitrary meshes into a strip-based representation (M. Deering, 1995; Evans, Skiena and Varshney, 1996; Chow, 1997). Hoppe (1999) introduced the idea of reordering vertex indices—relying on a post-transform cache in the graphics processor—to dynamically take advantage of locality of reference when possible, without requiring all geometry submitted for rendering to be encoded first. The specifics of cache operations are thereby abstracted to avoid sensitivity to differences in hardware architectures. Details on cache behavior of early GPU architectures can, howeve,r be found in corresponding documentation (Riguer, 2006), as well as hardware proposals and simulation frameworks (Sheaffer, Luebke and Skadron, 2004; P.-H. Wang et al., 2011).

Generalizing from triangle strips to arbitrary indexed triangle sets has enabled more recent work to consider highly elaborate reordering schemes. Lin and Yu (2006) describe the *K-Cache* algorithm, which includes a predictive simulation of the cache evolution in each iteration, rendering their approach quite time-consuming. Forsyth (2006) omits this predictive simulation for final cache positions, but assumes an exponential falloff for the probability of a cache hit instead and thus requires no parameterization. As a consequence, a low cache miss rate comparable to that of Lin and Yu can be achieved with significantly shorter required processing time. Its speed makes the algorithm by Forsyth a popular choice in the industry. Sander, Nehab and Barczak (2007) have presented another extremely fast, scalable algorithm named *Tipsify* for reordering vertex indices as part of their mesh optimization tool chain. *Tipsify* works particularly well with larger cache sizes, as has been confirmed in experimental evaluations.

Recent work on out-of-core geometry processing has shown how data can be rearranged without changing the mesh itself to improve performance even in applications beyond rendering (Isenburg and Lindstrom, 2005). Cache considerations may also be applied when processing or transmitting geometry. Chhugani and Kumar (2007) have explored the concept of cache-based optimization to achieve a compression-friendly topology representation. The authors report cache miss rates on par with those of Lin and Yu, as well as extremely low storage requirements of only ∼8 bits per triangle. Considering the entire cache hierarchy from hard drive to main memory to rendering system, appropriately reordering geometry data can significantly increase transfer rates (S. Yoon and Lindstrom, 2007; Tchiboukdjian, Danjean and Raffin, 2010; Tchiboukdjian, Danjean and Raffin, 2008).

While the methodologies, theoretical benefits and widespread adoption of the above approaches are rather impressive, evidence that the assumed post-transform cache is still part of modern hardware is virtually non-existent. Hence, an in-depth evaluation of cache-based optimization methods on recent GPU models is strongly needed to assess their contemporary usefulness. If we can, in fact, verify—as has been recently suspected—that the idea of a post-transform cache has become obsolete, we have reason to believe that other stages of the hardware pipeline may also not behave as expected.

2.2.2. Rasterization

In a sort-middle architecture, primitives that leave the geometry stage are commonly sorted into image-space bins, to enable independent pixel updates in these image regions by available shading units and reduce synchronization overhead. In addition to its application in hardware rasterization, subdividing the viewport into spatial bins or tiles has become common practice in the pursuit of high-performance software rendering (Seiler et al., 2008; Molnar, Eyles and Poulton, 1992; Clarberg, Toth and Munkberg, 2013; Patney, Tzeng et al., 2015). A notable example for a fully-programmable GPU software rasterization pipeline is presented by Laine and Karras (2011). However, in contrast to hardware rendering which follows a streaming design, they require that rendered geometry is preprocessed and the distribution of clip-space triangles is known before rasterization, which enables them to balance workload dynamically. In the hardware streaming pipeline, the process of assigning primitives to bins, which eventually map to processing cores, must occur ad-hoc and usually follows a static, built-in spatial pattern.

Previously suggested patterns for subdividing the screen space for parallel processing include using scanlines, horizontal and vertical strips or rectangular tiles (L. Wang et al., 2011). Juliachs, Carrard and Nominé (2007) shuffle 2D portions of the viewport and distribute them to available rasterizers randomly. M. W. Eldridge (2001) illustrates a tiling pattern for interleaved tile quads in the renowned *Pomegranate* architecture (M. Eldridge, Igehy and Hanrahan, 2000). However, the authors neither elaborate on how this pattern is produced, nor how its performance would be affected by scaling the tile size or the number of processors. Molnar, Cox et al. (1994) generally recommend using small bin sizes as a means to achieve better load balance. Naturally, as bins get smaller, work is more evenly distributed. A mathematical approach to predict performance curves and ideal bin sizes was presented by McManus and Beckmann (1996). However, decreasing bin size also implies an increase of the total workload itself, since triangles have to be processed by every bin they overlap. Thus, the choice of an appropriate binning strategy is a delicate one and plays a notable role in the design of a graphics pipeline (M. Chen et al., 1998; J. Chen et al., 2005; Dan Crişu, 2012).

The above material and publications, while originating from varied disciplines, provide us with extensive knowledge and a suitable scientific background to fulfill the objectives of this thesis. The expert knowledge of these esteemed authors enables us to make new assumptions, perform meaningful experiments and properly interpret the obtained results. We will revisit them at the appropriate times to complement their findings with our own insights.

3. Overview

Contents

3.1. Prioritized Dynamic Load Balancing on GPUs

In the first part of this thesis, we will focus on fully dynamic strategies for load balancing on the GPU, with additional emphasis on prioritization and applications to software rendering. The author has contributed to several papers on this topic that we list below and contextualize in Figure 3.1.

1. Markus Steinberger, Bernhard Kainz, **Bernhard Kerbl**, Stefan Hauswiesner, Michael Kenzel and Dieter Schmalstieg (Nov. 2012). 'Softshell: dynamic scheduling on GPUs'. In: *ACM Trans. Graph.* 31.6, 161:1–161:11

   The author has contributed the initial CUDA implementation for parallel progressive sorting of work items. He also wrote a substantial part of the path tracing and scene graph parsing applications used for evaluation.

2. Markus Steinberger, Michael Kenzel, Pedro Boechat, **Bernhard Kerbl**, Mark Dokter and Dieter Schmalstieg (Nov. 2014). 'Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU'. In: *ACM Trans. Graph.* 33.6

   The author has helped compare Whippletree with Softshell, tested early versions and implemented the REYES micropolygon rendering example.

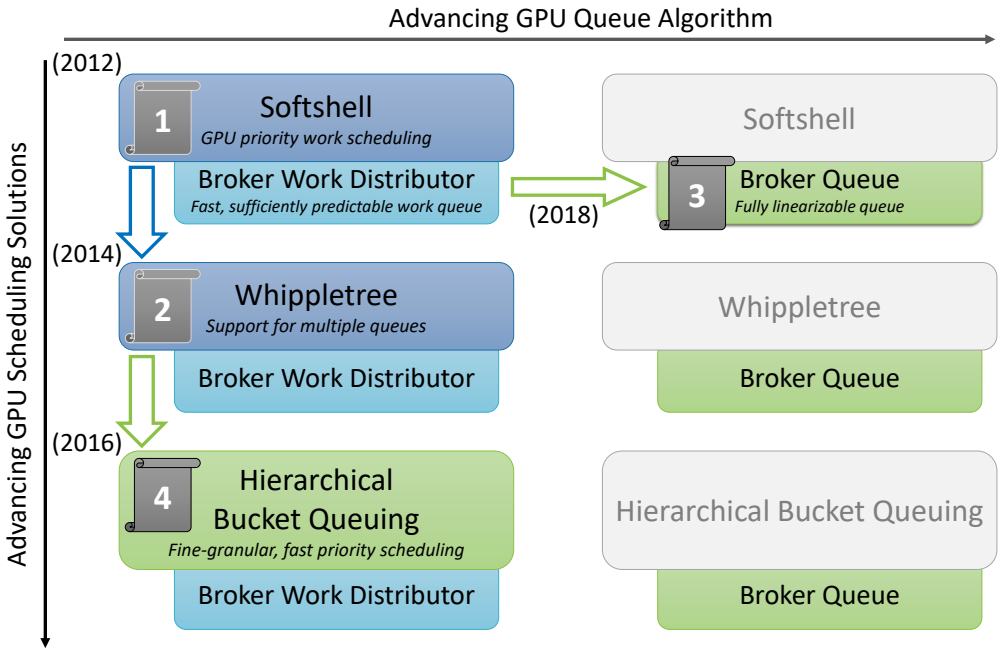


Figure 3.1.: Overview of relevant dynamic GPU scheduling solutions in context. The novel contributions that we present as part of this thesis are indicated by green boxes.

3. **Bernhard Kerbl**, Michael Kenzel, Joerg H. Mueller, Dieter Schmalstieg and Markus Steinberger (2018). 'The Broker Queue: A Fast, Linearizable FIFO Queue for Fine-Granular Work Distribution on the GPU'. In: *Proceedings of the International Conference on Supercomputing*. ICS '18. Beijing, China

   The author did most of the writing and analysis, comparison and practical applications of the broker queue. Joerg Mueller helped with the evaluation. Michael Kenzel and Markus Steinberger provided the initial broker work distributor algorithm. Dieter Schmalstieg revised the paper.

4. **Bernhard Kerbl**, Michael Kenzel, Dieter Schmalstieg, Hans-Peter Seidel and Markus Steinberger (2016). 'Hierarchical Bucket Queuing for Fine-Grained Priority Scheduling on the GPU'. In: *Computer Graphics Forum*

   The author wrote most of the paper and provided the hierarchy designs as well as adaptive rendering applications. Dieter Schmalstieg and Hans-Peter Seidel contributed their expertise. Bucket queuing is largely based on previous work by Michael Kenzel and Markus Steinberger.

In order to establish a frame of reference for the chapters on dynamic load balancing, let us compare the contributions in these publications and the individual tools for priority scheduling on the GPU presented therein.

Instead of relying on writing separate, light-weight kernels for performing different tasks in a pipeline, one can alternatively launch a larger, persistent megakernel that can take care of managing versatile tasks, in order to circumvent these restrictions. The *Softshell* framework occupies the entire GPU with such a persistent megakernel and makes threads continuously look for new work to process. The elemental key component in this approach is a parallel queuing data structure, from which work packages can be drawn. In contrast to previous approaches, Softshell enables persistent GPU threads to enqueue new work packages (or *items*), hence task consumption and creation is completely dynamic and can be autonomously scheduled on-chip. Additionally, Softshell enables the evaluation of custom priority functions for every item in the queue. A dedicated group of warps takes care of progressively sorting the queue based on these priorities at runtime, as shown in Figure 3.2.

The idea of dynamic scheduling on the GPU was investigated further in the creation of *Whippletree*, which demonstrated improved efficiency by exploiting shared device memory where possible and supporting different levels of parallelism for concurrent tasks. As with Softshell, queuing plays a central role in Whippletree, which provides queues in both global and shared memory for storing and fetching new work packages to be processed in a persistent threads megakernel. In order to minimize overhead, support for work prioritization was weakened by omitting the slow sorting routine and only allowing to define a fixed priority for all work items that perform the same procedures and are stored in the same queues. While significantly coarser than having fully dynamic priorities for each item, this strategy still enables Whippletree to process high-priority jobs before less important ones with great efficacy.

Although they form an essential part of both Softshell and Whippletree, their underlying queue structures have not been explicitly examined. Efficient queues for work distribution on the GPU pose a particular challenge due to the extreme level of parallelism they must support. Chapter 4 provides a state-of-the-art evaluation of concurrent queuing techniques, lists key properties for GPU execution and shows how they can be considered in the design of parallel queuing algorithms. We demonstrate this by example of the *Broker Work Distributor*, which we successfully employed in both Softshell and Whippletree, as well as the *Broker Queue*, which can serve as a linearizable alternative in any solution that uses the broker work distributor, as indicated in Figure 3.1.

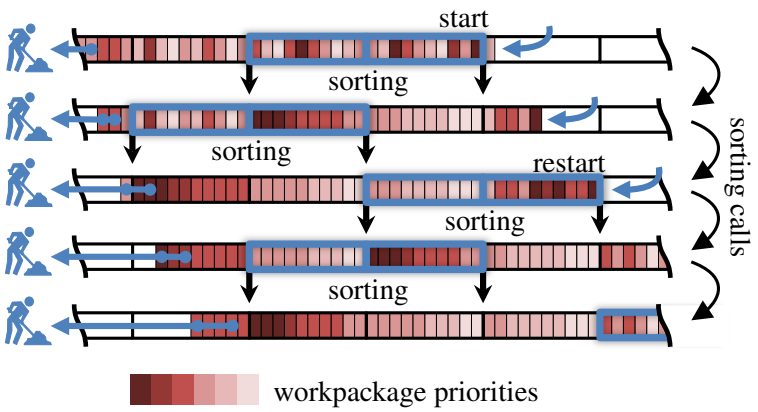


Figure 3.2.: When progressively sorting priority work queues, we must only modify segments that are safe to access and restart from the back if we get too close to the front.

Unfortunately, neither Softshell nor Whippletree can fully solve the problem of load balancing under prioritization. The rather sluggish sorting approach of Softshell only takes effect when executing work packages with exorbitant runtimes, giving the dedicated warps enough time to produce a reasonably sorted queue. Especially for graphics applications, which are often bound by real-time processing constraints, relying on such a concept is not an option. Procedure-based prioritization, as supported by Whippletree, is also not sufficient for graphics pipelines, as it cannot capture fine-granular significance of individual work items in a stage: although the process is the same, shading a triangle in close proximity to the camera may be vastly more important than one that is far away. In Chapter 5 of this thesis, we thus describe and evaluate *Hierarchical Bucket Queuing*, which enables quota-driven scheduling on the GPU and fine-granular prioritization fit for adaptive rendering. A comparison of key features with Softshell and Whippletree is provided in Table 3.1.

Table 3.1.: Properties and features of individual GPU priority scheduling frameworks.

	Softshell	**Whippletree**	**Hierarchical Bucket Queuing**
Queue(s)	One	Multiple	Many
Basic structure	Monolithic	List	Tree
Queue memory	Global	Global/Shared	Global/Shared
Work prioritization	Fine	Coarse	Fine
On-line queue sorting	Required	None	Optional
Runtime overhead	High	Low	Low

Much like Softshell and Whippletree, the priority scheduling solution presented in this thesis is based on launching a persistent threads GPU megakernel and relies on an efficient queuing algorithm to provide core functionality for task scheduling. Similarly, the active threads also interact with queues that are maintained on-chip to receive work and can submit new work items dynamically. In fact, our implementation of hierarchical bucket queuing was created by integrating various new concepts into the existing Whippletree framework, which has enabled us to exploit available core functions and facilitated a fair comparison with previous work. Since we built our solution on the Whippletree architecture, we can also use the same optimizations for global and shared memory queues to benefit from locality of reference.

In contrast to interfacing with a single monolithic queue in Softshell and Whippletree's basic multi-queue support, hierarchical bucket queuing targets designs that feature a significantly higher number of queues. In this context, a queue can be understood as an abstract container that implements methods for enqueuing and dequeuing work items. An invoked transaction can then be recursively forwarded into one of possibly many subjacent child queues, or *buckets*, until it reaches a terminal queue that performs the actual storage management. Since there is no limitation with regard to these parent-child relationships, our design encourages the creation of tree-like hierarchies with arbitrary complexity and makes it easy to deploy fine-granular work prioritization policies. Hierarchical bucket queuing, like Softshell, supports on-line sorting of work items in terminal queues, which, given enough time, can produce an accurately ordered priority queue with arbitrary granularity. However, this behavior is optional and only enabled on-demand in our implementation, since the combination of hierarchies and bucketing often suffices to bring about adequately prioritized processing of tasks.

By including a detailed description of a massively parallel queuing algorithm and a fast, intuitive priority scheduling approach, the contents of Chapters 4 and 5 convey the essentials for building elaborate load balancing solutions on the GPU. In contrast to a simple persistent threads approach, we can now build applications that exhibit adaptive behavior and focus processing power on particularly important tasks while maintaining a well-balanced compute core load. Thanks to the low runtime overhead of the Whippletree framework our implementation is based on, prioritization with hierarchical bucket queuing can satisfy soft real-time constraints and thus provides an adequate basis for creating adaptive software rendering applications, as we will demonstrate in the first part of this thesis.

3.2. Static Load Balancing for GPU Rendering

In the second part of this thesis, we shift our focus from dynamic load balancing to static variants in the context of GPUs as high-performance, sort-middle rasterization architectures. The benefits of doing so are three-fold: First, we gain new insights into state-of-the-art load balancing schemes and how they suit the design of modern GPUs. Second, from these findings, we can infer steps to optimize input data to best exploit the prevalent hardware load balancing mechanisms. Third, we can consider them as a lodestar for deriving suitable and effective load balancing policies for parallel sort-middle software rendering, tailored towards effective execution on the GPU architecture.

The corresponding chapters of this thesis can (and should) be understood in the context of a larger project that the author was involved in, namely the development of a fully programmable software rendering pipeline. Research into this topic has been rewarded by a co-authored publication:

5. Michael Kenzel, **Bernhard Kerbl**, Dieter Schmalstieg and Markus Steinberger (Nov. 2018). 'A High-Performance Software Graphics Pipeline Architecture for the GPU'. In: *ACM Trans. Graph.* 37.4

   The author provided the CUDA implementation for deploying the sort-middle assignment of primitives to screen-space bins. He also wrote most of the applications, helped with collecting meaningful test results and writing the paper.

The main contribution of this paper features a complete DirectX 9-style streaming pipeline for rendering, written entirely in C++/CUDA, and hence dubbed *CUDA rendering engine*, or CURE for short. Its aim is to take advantage of the parallel processing power of the GPU to perform high-speed rendering tasks, while still preserving full programmability at every pipeline stage. In contrast to previous GPU software rendering approaches, the streaming design enables data flow control as well as running multiple pipeline stages concurrently, thus avoiding device-wide synchronization and memory consumption spikes. Figure 3.3 outlines the general design of the basic CURE rendering pipeline. Note that CURE, like the dynamic load balancing methods discussed above, is dependent on a concurrent queuing solution.

The fundamental operating principles of the hardware rasterization pipeline are well-known, and understanding them poses a prerequisite for creating elaborate real-time graphics applications against available rendering APIs.

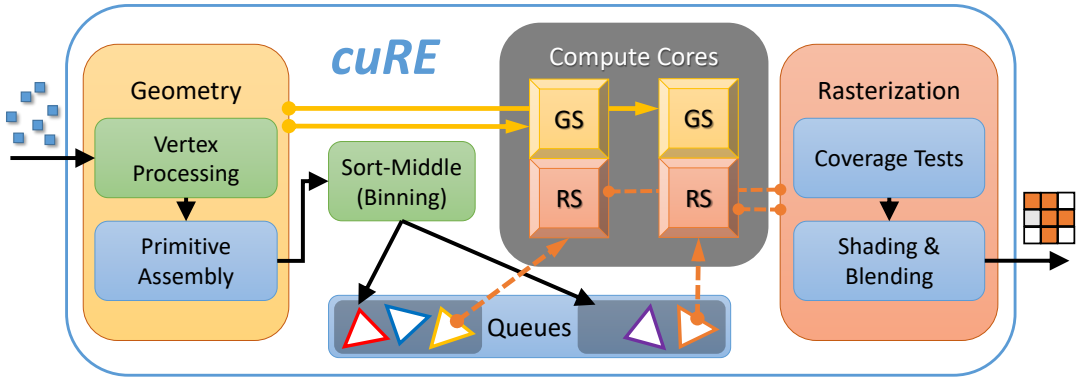


Figure 3.3.: The CURE streaming pipeline concept. In contrast to other software pipelines, data can be transferred between stages without requiring multiple kernel launches. The compute cores can perform tasks from the geometry stage (GS) or rasterization stage (RS) concurrently, exploiting shared GPU memory for organizing workload within stages. Crossing from GS to RS, primitives have to be sorted and stored in designated queues that map to groups of compute cores with exclusive access to a portion of the framebuffer. After completing the rasterization tasks, the output can be written without interference or having to rely on global synchronization.

However, these conventional APIs conceal the internal, integral load balancing mechanisms that are required to ensure that the available parallel compute power is optimally utilized, as data transfers from one stage to the next. In the process of conceiving a high-performance software solution for the GPU that closely mimics the hardware pipeline, these questions suddenly become highly relevant. In our research, we have identified concrete solutions for load balancing problems in sort-middle rendering by uncovering undocumented GPU scheduling policies, designing meaningful experiments and verifying anticipated performance benefits for hardware and software rendering. These achievements arose naturally during the implementation of the overarching CURE project. Specifically, the chapters on static load balancing will address the findings and solutions we obtained while working on the particular pipeline stages enclosed in Figure 3.3 by green boxes.

In Chapter 6, we tackle the widely accepted preconception regarding the existence of a post-transform cache for vertex reuse in hardware rendering. We disprove the existence of a centralized cache on recent models from three major GPU vendors. Our investigations indicate how vertex information is actually reused during rendering and reveals the underlying load balancing schemes.

Based on these insights, we devise an algorithm for mesh optimization which reorders vertex indices to maximize reuse and minimize vertex shader invocations. The majority of this content is based on a previous publication:

6. **Bernhard Kerbl**, Michael Kenzel, Elena Ivanchenko, Dieter Schmalstieg and Markus Steinberger (Aug. 2018). 'Revisiting The Vertex Cache: Understanding and Optimizing Vertex Processing on the modern GPU'. In: *Proc. ACM Comput. Graph. Interact. Tech.* 1.2

   The paper was written entirely by the author, who also contributed to the reverse-engineering process and provided baseline implementations for optimization algorithms. Michael Kenzel wrote the DirectX 12 batch interpreter and Elena Ivanchenko developed tools for batch visualization and evaluation. Markus Steinberger contributed the implementation for batch prediction and batch-based optimization. Dieter Schmalstieg helped universally with his expert knowledge and guidance.

Another vital application of static load balancing for rendering is examined in Chapter 7, where we consider the topic of binning for rasterization. We identify how the assignment of primitives to individual processing units follows particular patterns and, based on statistical analysis, present alternative approaches that scale well with increasing processor counts. While these findings may not necessarily affect decisions for future hardware designs, we show that their consideration can significantly raise performance for state-of-the-art software rasterization. The chapter is based on a conference paper:

7. **Bernhard Kerbl**, Michael Kenzel, Dieter Schmalstieg and Markus Steinberger (2017). 'Effective Static Bin Patterns for Sort-middle Rendering'. In: *Proceedings of High Performance Graphics*. HPG '17. Los Angeles, California: ACM

   The author did most of the paper writing and designed all experiments, visual material and auxiliary tools required for evaluation. Michael Kenzel and Markus Steinberger helped with the writing and provided the initial motivation for this research. Dieter Schmalstieg oversaw the creative process and helped improve the paper on several occasions.

3.3. Further Publications

The following publications indicate additional research projects that the author was involved in during his doctoral studies. Some of them, while representative of his scientific interests, do not fit the theme of this thesis and were therefore omitted. For others, although they may be of significant relevance to the discussed topics, their main contribution cannot be attributed to the author. However, we will highlight their importance and influence on the formation process for this thesis where appropriate.

8. **Bernhard Kerbl**, Philip Voglreiter, Rostislav Khlebnikov, Dieter Schmalstieg, Daniel Seider, Michael Moche, Philipp Stiegler, R. Horst Portugaller and Bernhard Kainz (2013). 'Intervention Planning of Hepatocellular Carcinoma Radio-Frequency Ablations'. In: *Clinical Image-Based Procedures. From Planning to Intervention*. Vol. 7761. Lecture Notes in Computer Science. Springer Berlin Heidelberg

   The author wrote the entire paper and designed the user interface for initiating different established simulation and planning algorithms by the co-authors.

9. **Bernhard Kerbl**, Denis Kalkofen, Markus Steinberger and Dieter Schmalstieg (2015). 'Interactive Disassembly Planning for Complex Objects'. In: *Computer Graphics Forum* 34.2

   The entire paper, implementation of the code base and evaluation was done by the author. Denis Kalkofen, Markus Steinberger and Dieter Schmalstieg provided their expertise during the initial conception and revision of the paper.

10. Peter Mohr, **Bernhard Kerbl**, Michael Donoser, Dieter Schmalstieg and Denis Kalkofen (2015). 'Retargeting Technical Documentation to Augmented Reality'. In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. CHI '15. Seoul, Republic of Korea: ACM

    The author provided his knowledge and available code for motion planning in reproducing disassembly instructions. He developed parallel implementations for most integral subprocedures. He also assisted in paper writing, algorithm evaluation and the collection of test results.

11. **Bernhard Kerbl**, Joerg H. Mueller, Michael Kenzel, Dieter Schmalstieg and Markus Steinberger (2018). 'A Scalable Queue for Work Distribution on GPUs'. In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '18. Vienna, Austria: ACM

    This publication represents an early, shortened version of the later accepted full paper on the broker queue. The contributions by the authors are therefore identical.

12. Michael Kenzel, **Bernhard Kerbl**, Wolfgang Tatzgern, Elena Ivanchenko, Dieter Schmalstieg and Markus Steinberger (Aug. 2018). 'On-the-fly Vertex Reuse for Massively-Parallel Software Geometry Processing'. In: *Proc. ACM Comput. Graph. Interact. Tech.* 1.2

    A significant portion of the paper writing process was organized, supervised and supported by the author. He also helped with the implementation of the application for computing mesh envelopes and further collected and interpreted experimental results.

4. The Broker Queue

Contents

Before we begin to dissect the existing dynamic GPU load balancing strategies and try to improve them with high-level concepts, we first need to revisit the fundamental building block of GPU scheduling: concurrent queuing. Detailed knowledge on how to build a fast, versatile and reliable parallel work queue not only gives us a solid foundation for developing novel load balancing approaches on top of it, but can, in fact, benefit any application that seeks to circumvent the limit of rigid, built-in GPU kernel scheduling.

4.1. GPU Scheduling & Concurrent Queues

At the core of most dynamic scheduling strategies for GPUs are concurrent queues, which collect and distribute work, usually in a FIFO manner (Cederman and Tsigas, 2008; Steinberger, Kainz et al., 2012; Steinberger, Kenzel, Boechat et al., 2014). The available literature on concurrent queues has a strong focus on lock-freedom, which is often held as key to performance in concurrent systems. However, these algorithms are commonly geared towards CPU architectures and do not cater to the peculiarities of powerful and ubiquitous GPU hardware. The lock-free property is often achieved by methods in the spirit of *optimistic concurrency control* (Kung and Robinson, 1981), *e.g.*, through algorithms that assume a low incidence of failed atomic operations from competing threads. However, as others (M. Herlihy, Luchangco and Moir, 2003; Hendler, Incze et al., 2010) have already noted, the overhead that is actually caused by repeatedly failing code sections can outweigh the benefits of true lock-freedom. As an alternative, blocking queues have been proposed specifically for the GPU (Scogland and Feng, 2015). Unfortunately, conventional blocking queues substantially limit options for load balancing, as they do not return the control to the calling thread in underflow or overflow situations. Thus, they cannot be used in multi-queue setups and advanced work distribution strategies, *e.g.*, work stealing (Chatterjee et al., 2011).

In this chapter, we present a new queue design, fit for work distribution and general queuing on the GPU. First, we identify desired properties for efficient work distribution on the GPU and assess the fitness of previous algorithms in this respect (Section 4.2). Based on these properties, we describe a scalable, linearizable queue, the *broker queue* (BQ), which shows the performance of a blocking queue, but can return control to the scheduler in case of underflow or overflow (Section 4.3). Additionally, we present two variants of the BQ, which can further improve performance at the expense of linearizable FIFO behavior. All presented algorithms

- support all execution paradigms of the GPU: individual threads, thread groups of SIMD width (warps), and cooperative thread groups (blocks),
- store items in a ring buffer and thus avoid costly memory allocation,
- ensure that enqueue/dequeue are not fully-blocking, if the queue is full or empty, thus enabling multi-queue setups, and
- only show blocking behavior for threads if they try to interact with the same queue entries, to avoid read-before-write hazards, limiting blocking conditions to cases where it is absolutely necessary.

We prove linearizability of the broker queue (Section 4.4) and describe specifics for implementing variants of BQ in Section 4.5. We compare our designs to the state-of-the-art in both synthetic tests and realistic use cases (Section 4.6).

4.2. Requirements for Massively Parallel Queues

Queuing algorithms on the GPU not only have to handle thousands of concurrent enqueue and dequeue operations correctly, they also need to consider the specifics of the underlying hardware. This includes confinement to a limited amount of memory, constraining register usage, and operating on a SIMD device, where individual lanes can diverge. Thus, the requirements for an efficient work queue on the GPU differ significantly from those on the CPU.

We will compare our insights with an extensive body of previous algorithms and categorize the most relevant techniques with regard to these desired properties: An early array-based queue (GQ) by Gottlieb, Lubachevsky and Rudolph (1983); A popular lock-free queue (MSQ) by Michael and Scott (1996) and a blocking queue by the same authors using two locks, which we call *dual mutex queue* (2MQ); The first practical lock-free, array-based queue (SHCQ) by Shann, Huang and C. Chen (2000); A queue by Tsigas and Zhang (TZQ), which updates its contents periodically (2001); The baskets queue (BAQ) by Hoffman, Shalev and Shavit (2007); A circular buffer queue (CBQ) and its high-throughput alternative (HTQ) by Orozco et al. (2012); The lock-free linked concurrent ring queue (LCRQ) by Morrison and Afek (2013), consisting of multiple smaller, concurrent ring queues (CRQs); A fast blocking queue (SFQ) by Scogland and Feng (2015) and its non-blocking variant (NSFQ). And finally, a recently proposed wait-free queue (WFQ) devised by Yang and Mellor-Crummey (2016); A comprehensive listing for availability of identified requirements in the above queuing methods, as well as our technique, is available in Table 4.1.

Most recent non-blocking queuing algorithms rely on optimistic concurrency control (Kung and Robinson, 1981). However, the high resource contention on the GPU—when thousands of threads try to access the same data element—can lead to a significant number of retries, *e.g.*, hundreds to thousands of repeated compare-and-swap (C&S) operations for a single enqueue. Obviously, such behavior impacts performance negatively. In accordance with other authors (M. Herlihy, Luchangco and Moir, 2003; Hendler, Incze et al., 2010), we argue that in order to design a work queue that is ***highly scalable***,

a potentially blocking algorithm is preferable over using contended C&S operations. To avoid retries on failed C&S operations, every thread has to be assigned a unique spot in the queue. This requirement intuitively leads to an array-based queue design, using atomic fetch-and-add (F&A) to increment front and back pointers (or *head* and *tail*) of the queue instead of C&S. Even though such a design also brings forth two points of contention, performance on the GPU can still be high, as F&A operations are exceptionally efficient on recent hardware generations (Harris, 2014).

Using F&A on front and back pointers, in combination with a per-element ticketing system, can be further extended to enable ***fair ordering***: the algorithm does not constrain head and tail pointers to the size of the ring buffer, but rather allows them to wrap around to simulate an array of infinite length, so they can yield both a ring buffer location and a ticket. As soon as a thread reaches the F&A on the pointer, its position in the queue is assigned. From that point on, there is no risk of that thread taking significantly longer to complete an enqueue or dequeue due to interference of queuing-related operations, *e.g.*, due to failing C&S. Our queue, CBQ and SFQ offer these guarantees; LCRQ and WFQ use tickets that may be invalidated by contending threads. Another important property of a queue for work distribution is guarantee of predictable behavior. For example, the queue must not sporadically report over-/underflow or have queued elements seemingly change order. The accepted standard for proving predictable behavior is ***linearizability***, which applies to most related work except for GQ, as shown by Blelloch et al. (2003), HTQ (Orozco et al., 2012), and TZQ, as shown by Colvin and Groves (2005).

On the GPU, the degree of parallelism (or *occupancy*) that can be achieved at runtime is dictated by the resource requirements of a kernel. For example, exceeding a certain number of registers may reduce the number of concurrently launched warps and thus the ability of the GPU to effectively hide latency. Since the queuing algorithm must be embedded in the kernel in order to use it for work distribution, a ***low resource footprint*** is desirable to allow for high occupancy of the routines built on top of it. Due to their sophisticated design, even bare-bone implementations of LCRQ and WFQ reduce achievable occupancy on current GPU models according to our tests.

Large numbers of dynamic memory management operations are known to be the cause of potential bottlenecks for GPU execution (Steinberger, Kenzel, Kainz et al., 2012). Using ***static memory only*** implicitly avoids these potential overheads. Hence, a work queue on the GPU should avoid dynamic memory allocation, which, in theory, gives array-based queues a technical advantage

Table 4.1.: While many parallel queues have been proposed, most lack desired properties for efficient work distribution on the GPU. General lock-free queues are commonly dependent on dynamic memory and therefore difficult to realize on the GPU. Faster queuing approaches either lack linearizability, or their rigorous blocking behavior precludes multi-queue setups. Our queue (BQ) fulfills all identified desired properties for massively parallel work distribution.

	GQ	MSQ	2MQ	SHCQ	TZQ	BAQ	CBQ	HTQ	LCRQ	SFQ	NSFQ	WFQ	**BQ**
highly scalable	●						●	●	●	●		●	●
fair ordering							●		○	●		○	●
linearizability		●	●		●	●	●		●	●	●	●	●
low resource footprint	●	●	●	●	●	●	●	●		●	●		●
static memory only	●		●							●	●		●
all execution paradigms	●	●	●	●	●	●		●	●		●	●	●
multi-queue support	●	●	●	●	●	●		●	●		●	●	●
multi-element dequeue			●										●

● . . . fulfills the requirement, ○ . . . partially fulfills the requirement

over common linked-list alternatives. However, array-based queues often still need to allocate memory for queued elements individually, as the queue storage is operated using C&S, and thus can only store pointers to the actual elements. If elements are instead stored in the queue directly, access to it needs to be secured, to avoid read-before-write and write-before-read hazards.

The design of the GPU architecture yields multiple ***programming and execution paradigms***. General queue designs must be able to work within all of them, including independent thread execution, warp-synchronous execution, sub-warp execution, and cooperative block execution. This requires a queue design that does not transfer blocking states between threads in the same warp, *i.e.*, block ready-to-execute threads, as others are stalled in the queue. Similarly, ***multi-queue*** setups require threads to return from dequeue operations, if a queue is already empty, so they can probe other queues that might still hold work. Essentially, both requirements boil down to queues being non-blocking when a queue is full or empty.

Our proposed design, the broker queue—although forgoing the non-blocking property of most recent queue designs—exhibits all desired properties listed above. Furthermore, it enables a thread to ***dequeue multiple elements*** at once, raising efficiency in cooperative block execution scenarios. It shows all advantages of simpler, conventional blocking queues, while also ensuring linearizability and detecting overflow and underflow without blocking.

4.3. The Broker Queue

The core functionality of the broker queue is defined by its four integral components: (1) a ring buffer that can store queued elements directly, (2) a head and a tail pointer for ticketing, (3) a ticket buffer that locks individual queue elements, and (4) an explicit counter to weigh enqueue against dequeue operations. The configuration of these buffers and the interface to enqueue/dequeue is given in Algorithm 1. Note that $\Leftarrow\!\!\!|$ indicates an atomic transaction, whereas $\Leftarrow$ is a memory access and $\leftarrow$ a local variable assignment.

The key characteristic of this queue interface can be viewed as a *broker*, giving rise to the name of our queue. The broker not only considers items stored in the ring buffer of the queue: it also accepts assurances to provide or consume items, before the actual transactions occur.

ALGORITHM 1: Broker Queue of size **N**

```
1  QueueElements RingBuffer[N]                with N = 2^n
2  unsigned int Tickets[N] ← {0, 0, ··· , 0}
3  unsigned int Head ← 0, Tail ← 0
4  int Count ← 0
5  enqueue (Element)
6      while not ensureEnqueue () do
7          (head, tail) ⇐ (Head, Tail)
8          if N ≤ tail − head < N + MaxThreads/2 then
9              return Full
10     putData (Element)
11     return Success
12 ensureEnqueue ( )
13     Num ⇐ Count
14     while true do
15         if Num ≥ N then
16             return false
17         if atomicAdd (Count,1) < N then
18             return true
19         Num ← atomicSub (Count,1) −1
20 putData (Element)
21     Pos ← atomicAdd (Tail,1)
22     P ← Pos % N
23     waitForTicket (P, 2 · (Pos/N))
24     RingBuffer[P] ⇐ Element
25     Tickets[P] ⇐ 2 · (Pos/N) + 1
26 dequeue ( )
27     while not ensureDequeue () do
28         (head, tail) ⇐ (Head, Tail)
29         if N + MaxThreads/2 ≤ tail − head − 1 then
30             return Empty
31     return readData ()
32 ensureDequeue ( )
33     Num ⇐ Count
34     while true do
35         if Num ≤ 0 then
36             return false
37         if atomicSub (Count,1) > 0 then
38             return true
39         Num ← atomicAdd (Count,1) +1
```

```
40 readData ( )
41     Pos ← atomicAdd (Head,1)
42     P ← Pos % N
43     waitForTicket (P, 2 · (Pos/N) + 1))
44     Element ⇐ RingBuffer[P]
45     Tickets[P] ⇐ 2 · ((Pos + N)/N)
46     return Element
47 waitForTicket (Pos, ExpectedTicket)
48     Ticket ⇐ Tickets[Pos]
49     while Ticket ≠ ExpectedTicket do
50         backoff ()
51         Ticket ⇐ Tickets[Pos]
```

4.3.1. Brokering

Usually, atomically operated head and tail pointers for ticketing prohibit a non-blocking reaction to over- and underflow. For example, if the queue holds a single element and multiple threads increase the head pointer atomically, the head is moved past the tail. Although threads could detect that the pointer was moved too far, reverting the move is difficult, as it would require a coordinated effort of all involved threads. Additionally, other threads could, in the meantime, enqueue new elements, validating some of the dequeue operations that were already rolled back.

To avoid these issue, we introduce an additional counter variable (*Count*). It ensures that only threads which are guaranteed to eventually complete their enqueue or dequeue operation (and thus validly move head or tail) are allowed to interact with those pointers. For `enqueue`, this assurance is provided by the `ensureEnqueue` method, which returns **true**, iff there is either sufficient space in the ring buffer to store an element, or a sufficient number of other threads have already committed to dequeue elements. Similarly, `ensureDequeue` returns **true**, iff there is an element in the ring buffer for the thread to dequeue, or at least one other thread committed to enqueue an unclaimed element. Thus, *Count* essentially models the relation between head and tail after all operations of concurrently active threads have completed. If *Count* is decreased below zero or increased above the ring buffer size, a thread can perform a rollback with an inverse operation (lines 19 and 39), without the need of explicit coordination with other threads. Due to the possibility

of other threads modifying *Count* in the meantime, the result of the rollback may suggest that the operation is now, in fact, possible. As other threads may have picked up on this (previously invalid) assurance, the thread must try to verify it by atomically modifying *Count* one more time. This retry behavior requires a loop over the corresponding instructions (lines 13-19 and 33-39).

4.3.2. Data Storage and Exchange

The internal workings of the broker queue match the assurances of the broker to actual ring buffer slots and create a connection between `enqueue` and `dequeue` operations. A slot identifies the location for writing/reading to/from the centralized ring buffer storage of the broker queue. The atomic operations on *Head* and *Tail* (line 21 and 41) return a tally for computing the ticket number and, implicitly, a ring buffer slot for reading or storing elements (line 22 and 42). The ticketing itself assigns even-numbered tickets to enqueue operations and odd numbered tickets to dequeue operations. Since the broker has already confirmed at this point that performing the assured operations will yield a valid queue state and thus will eventually succeed, `putData` and `readData` simply implement a blocking behavior. This is achieved by waiting on a spinlock in `waitForTicket`, until the thread's turn has come to interact with the assigned ring buffer location. To achieve favorable scheduling, threads back off after an unsuccessful spin. Each successful operation increases the ticket number for a slot by the total ring buffer size. Consistency on integer wrap-around can be easily guaranteed by choosing a power of two for the size of the queue **N** and using unsigned integers for pointers and tickets.

The methods `ensureEnqueue` and `ensureDequeue` are themselves called from a loop by the queue interface. The motivation behind this design is linearizability. A broker state indicating no available slot does not necessarily guarantee that a **Full** or **Empty** must actually be observable in the linearized operation of the queue. For example, a thread might reduce the *Count* variable to zero through `dequeue`, but get suspended before changing the *Head*. Another thread—just examining the *Count* variable—would assume the queue to be empty, although the previously assured `dequeue` might happen much later, and thus the queue never (observably) reached the **Empty** state. As *Count* might have changed during the execution of `putData` or `readData`, threads are required to continuously try to register their operation. Therefore, a thread that detects a potential **Full** or **Empty** state waits until that state can be definitely observed (loop from line 6 to 9 and 27 to 30).

4.3.3. Further Remarks

Next to enabling simultaneous access by an arbitrary number of threads, the biggest advantage of the BQ is that threads are only stalled if the queue is close to running empty or full. If there is sufficient time between adding an element and it being read, no thread has to wait. Another advantage of the BQ is that the ticketing system can grant threads access to queue elements for an extended period. As read and write operations on the actual elements do not need to be atomic, the queue can return a pointer to the acquired slot, *i.e.*, returning *P* instead of reading or writing (line 24 / 44).

To determine whether the queue is full or empty, we rely on comparing *Head* and *Tail*. Thus, both variables need to be read in a single atomic instruction, which is enabled on current GPU designs by defining them as 32-bit wide offsets from the buffer address and placing them together in a 64-bit word. Alternatively, actual 64-bit pointers could be used on architectures that support the atomic C&S$_2$ operation used by Morrison and Afek (2013). Note that, due to our assurance-based interaction with the pointers, *Head* can overtake *Tail*, and the distance between the pointers can grow beyond the queue size. Thus, special care needs to be taken when comparing the pointers (line 8 and 29).

At first glance, it would appear that the *Count* variable presents a central choke point for the queuing algorithm. Recent approaches, such as LCRQ and WFQ, take special care to avoid singular, global variables for communicating queue states across threads. This is motivated by the fact that, on many conventional architectures (*e.g.*, x86), contended atomic operations incur a severe performance penalty. However, due to their importance for massively parallel applications, atomic operations are extremely efficient in GPU hardware and handle contention well. Figure 4.1 shows the average time required for F&A operations on a single global variable, relative to uncontended memory access. While this ratio rises sharply for CPU architectures with an increasing number of contending threads, the GPU architecture is much more forgiving. Furthermore, the contention on *Count* becomes significant only when the queue is facing either underflow or overflow; *i.e.*, when *Count* is changed multiple times by a single thread. Hence, the usage of *Count* in the algorithm comes at the consideration of the underlying hardware and its low demand in reasonably balanced enqueue/dequeue scenarios.

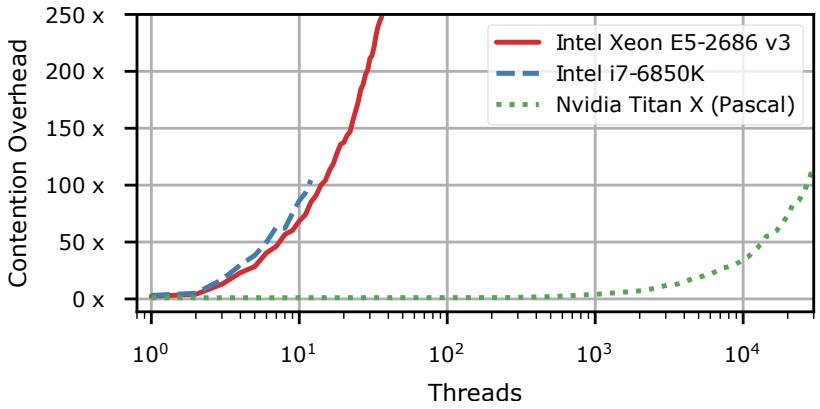


Figure 4.1.: Ratio of average time required for a contended F&A instruction to a single, non-atomic memory transaction on the respective architecture. On the GPU, a 10 000× contended F&A shows roughly the same overhead as 10× contention on the CPU.

4.4. Linearizability

To prove linearizability (M. P. Herlihy and Wing, 1990), one can model access to the queue as a history H. Every function call is represented by an invocation-response pair of events in the history. Two events are said to be ordered in H, if the response of one precedes the invocation of the other. If such an ordering is not possible for any two events, they are considered overlapping, and a linearizable data structure is allowed to order them arbitrarily. Linearizability is given if the partial ordering of event pairs can consolidate a total ordering such that the specifications of the data structure are fulfilled.

The semantics of the broker queue are those of a concurrent FIFO queue, which takes on three states: **Success** in case `enqueue` or `dequeue` succeeded, **Full** if `enqueue` is not possible, as the queue is full, and **Empty** in case there is no element left for `dequeue`. There are two relevant parts for showing linearizability of the broker queue: the exchange of data through `putData` and `readData`, as well as the brokering through `ensureEnqueue` and `ensureDequeue`.

4.4.1. Data Storage and Exchange

To show the linearizability of `putData` and `readData`, we consider threads that never see **Full** or **Empty** (ignoring `ensureEnqueue` and `ensureDequeue` for now). To this end, we use an auxiliary array H of infinite length, storing event pairs observed for every `enqueue` call $E_i = (e_i, \bar{e}_i)$ and `dequeue` call

$D_i = (d_i, \bar{d}_i)$. Each event shall be associated with its position in H, *i.e.*, $e_i < e_j$ iff e_i is recorded before e_j. An event shall be recorded during the atomic operations on *Tail* (e_i) and *Head* (d_i) (line 21 and 41) and after receiving a ticket ($\bar{e}_i$, $\bar{d}_i$) (line 23 and 43). For example, $H = \{e_1, e_2, \bar{e}_2, \bar{e}_1, d_1, \bar{d}_1, \dots\}$. Every event pair E_i and D_i shall be associated with $Pos = i$ obtained by the calling thread, and *Pos* shall reflect the FIFO ordering of elements in the queue (ignoring wrap-around of *Pos* for now). Thus, for linearizability, the following ordering must hold:

$$E_i < E_j \ \wedge \ D_i < D_j \ \wedge \ E_i < D_i \ \wedge E_i < D_j \quad \forall i < j$$

Obviously, $E_i < E_j$ and $D_i < D_j$ is trivial to observe, as the atomic counter makes sure that $e_i < e_j$ and $d_i < d_j$. Thus, either the respective calls are non-overlapping ($\bar{e}_i < e_j$, $\bar{d}_i < d_j$) and no reordering is necessary, or they do overlap and can be reordered to fulfill the requirements. For a single pair of calls E_i and D_i, it can be shown that $E_i < D_i$: given that tickets are unique, `waitForTicket` during `dequeue` must wait for `enqueue` to issue the `dequeue` ticket, and $\bar{e}_i < \bar{d}_i$. Thus, an ordering $E_i < D_i$ is certainly possible, as they are either ordered correctly or overlapping. What remains to be shown, is that all three requirements hold at the same time, *i.e.*, one reordering does not contradict another and $E_i < D_j \ \forall i < j$. The only possibility for an overall reordering to fail is if $\bar{d}_j < e_i$, *i.e.*, a `dequeue` finishes before an earlier `enqueue` starts, as this would make the calls non-overlapping and prohibit a reordering. This is not possible, due to the atomic operation on *Tail*, which yields $e_i < e_j$. In combination with $\bar{e}_i < \bar{d}_i$ and $e_i < \bar{e}_i$, we find $e_i < e_j < \bar{e}_j < \bar{d}_j$, and $\bar{d}_j \not< e_i$. Thus, all calls can be reordered according to *Pos*.

Since the ring buffer is of limited size, threads may potentially be competing to access the same slots. If multiple `enqueue` and `dequeue` operations are assigned to the same position, the ticket system makes sure that the order is kept as intended. The ticket for E_i is given by $T_{E_i} = 2 \cdot \lfloor i/\mathbf{N} \rfloor$, for D_i, $T_{D_i} = 2 \cdot \lfloor i/\mathbf{N} \rfloor + 1$. After a wrap-around, $T_{E_{i+\mathbf{N}}} = 2 \cdot \lfloor (i+\mathbf{N})/\mathbf{N} \rfloor = 2 \cdot \lfloor i/\mathbf{N} \rfloor + 2$ and $T_{D_{i+\mathbf{N}}} = 2 \cdot \lfloor i/\mathbf{N} \rfloor + 3$, *i.e.*, every operation receives a unique ticket which is monotonically increasing. In this way, the ordering at each spot of the ring buffer is ensured, as long as the tickets do not wrap around. If *Pos* wraps around at 2^{32}, the tickets wrap around at $2^{32}/\mathbf{N} = 2^{32-n}$. As long as the number of threads concurrently interacting with the queue stays below this value, the same ticket cannot be issued more than once at the same time. Hence, the order of operations on individual buffer slots follows *Pos*, and the queue in general maintains the indented linearizable FIFO behavior.

4.4.2. Brokering

Brokering evolves around the `ensure` functions, which may return **Full** or **Empty**. If `ensureEnqueue`/`ensureDequeue` returns **true**, a thread is forwarded to `putData`/`readData`, which results in linearizable behavior as outlined above. Thus, only the **Full** and **Empty** cases require a more detailed analysis. Ignoring the wrap-around of *Head* and *Tail* for now, we define two additional events $\infty_{h,t}$ and $\varnothing_{h,t}$. If the call returns **Full** or **Empty**, these events shall be respectively recorded during the combined head and tail reads (line 7 and 28), with $h = head$ and $t = tail$. Linearizability at underflow is given, if calls can be reordered such that an **Empty** state is reached at $\varnothing_{h,t}$:

$$D_t < \varnothing_{h,t} < E_{t+1}.$$

Ignoring wrap-around, **Empty** is returned for $t - h \leq 0$, *i.e.*, when both pointers are the same or *Head* has overtaken *Tail*. Observing such a pointer pair means that e_t and $d_t, d_{t+1}, \ldots, d_h$ have been recorded, and e_{t+1} has not happened yet:

$$e_t, d_t < \varnothing_{h,t} < e_{t+1}.$$

All D_i with $i > t$ are irrelevant for the **Empty** in question, all e_i and d_i for $i \leq t$ have already taken place (and thus E_i and D_i are either completed or overlapping), and e_{t+1} has not occurred yet. Thus, there is no E or D that prevents a reordering to achieve $D_t < \varnothing_{h,t} < E_{t+1}$. Furthermore, there are no other **Empty** or **Full** events that can interfere with creating such an **Empty** state: $\infty_{h,t}$ cannot take place at the same time (as the conditions for h and t are different). Another event $\varnothing_{h_2,t_2}$ with $t_2 = t$ and $h_2 = h$ may take place at the same time—and can simply be inserted right before or after $\varnothing_{h,t}$. An **Empty** event with $t_2 = t$ and $h_2 \neq h$ is also possible, which would be treated identically. If $t_2 < t$, the event has already been inserted into H earlier. Thus, the **Empty** state is linearizable. Linearizability with regard to the **Full** state is analogous to the **Empty** state, with $E_{h+\mathbf{N}} < \infty_{h,t} < D_{h+1}$, hence we omit repeating the derivation here.

Finally, the wrap-around of the pointer after 2^{32} must be considered. It is possible for *Head* to overtake *Tail*, with a factor equal to half the maximum number of concurrently active threads—if all threads are concurrently enqueuing and dequeuing, and all operations on the *Head* occur before the ones on *Tail*. Similarly, *Tail* can advance by half the maximum number of concurrently active threads further than **N** away from *Head*. These conditions can simply be included into the comparison as an additional margin (line 8 and 29). This condition obviously fails if $\mathbf{N} + MaxThreads/2 \geq 2^{32}$.

4.5. Broker Queue Variants

To ensure linearizability, our broker queue potentially waits until suspected **Full** and **Empty** states are observable from *Head* and *Tail*. Obviously, waiting comes at a cost. Hence, we also derive a simplified version of BQ which avoids waiting by shedding linearizability, yielding the *broker work distributor* (BWD). Dropping linearizable FIFO behavior opens the door for potentially even more efficient work distribution methods, *e.g.*, work stealing (Arora, Blumofe and Plaxton, 1998; Hendler, Lev et al., 2006). As BQ is also applicable in these use cases, we additionally describe the *broker stealing queue* (BSQ) for effective stealing of queued tasks.

4.5.1. The Broker Work Distributor

The conversion from broker queue to the broker work distributor is straightforward. Instead of waiting for `ensureEnqueue` and `ensureDequeue` in a loop to ensure **Full**/**Empty** are actually observable, these functions are called only once by `enqueue` and `dequeue`. The result of this call is taken at face value, returning **Full**/**Empty** if the broker cannot find a slot/match immediately. The downside of the BWD is its non-linearizability. Since *Count* is only used as an assurance swap, it does not faithfully represent the real queue state observable when the actual data is put into the queue or taken out of the queue. While this behavior is undesirable when a queue needs to behave strictly like a concurrent FIFO queue, it is not detrimental during work distribution.

If an `ensureEnqueue` yields **false**, it indicates that, according to all threads that started interacting with the queue thus far, all elements will be drained from the queue; *i.e.*, unless another thread starts `enqueue`, the queue will reach **Empty**. This behavior is arguably sufficient for work distribution and, with regard to multi-queue setups, provides a reasonable indicator for efficiently switching to another queue that might contain work.

4.5.2. The Broker Stealing Queue

The broker stealing queue (BSQ) provides a simple work stealing implementation by abstracting multiple underlying queues through one interface. Each executing block on the GPU is assigned its own, default BQ for storing and reading queued elements. If a thread in a block cannot find an item in its

assigned default queue, it tries to steal work from a different block. This is achieved by iterating over all available queues and performing a standard `dequeue` on each, until an element is found or all queues were checked.

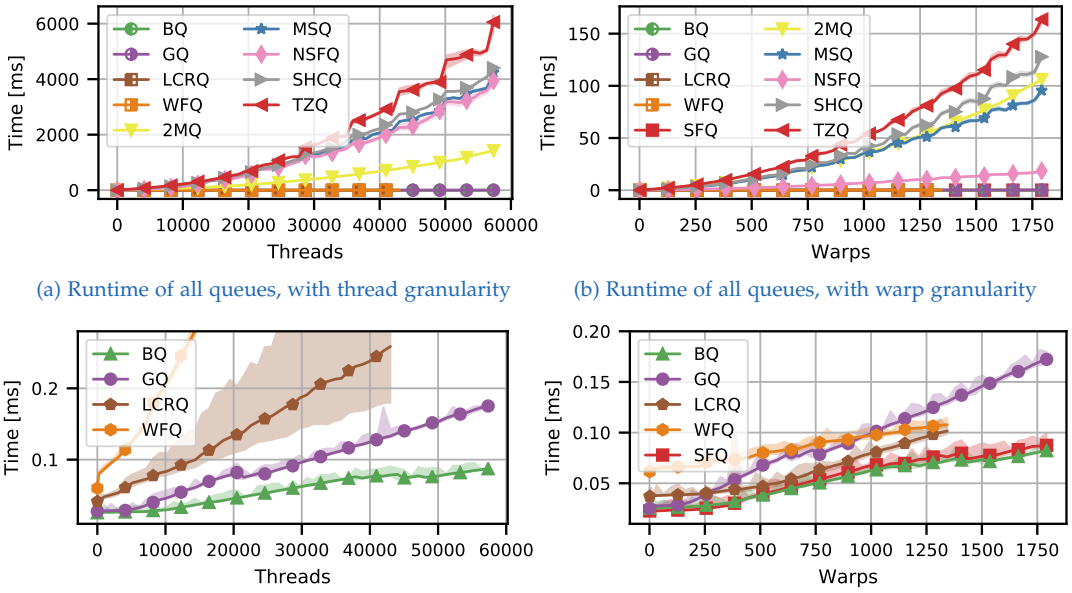
4.6. Evaluation

To evaluate our techniques, we compare their aptitude for work distribution with previous work. We implemented the queues listed in Table 4.1 in CUDA—with the exception of BAQ (as it is within 2× of MSQ), and CBQ and HTQ, which are similar to SFQ and GQ, respectively. In order to offer an exhaustive, yet reasonably concise evaluation of our algorithm against numerous previous approaches, we first identify the most competitive techniques in a microbenchmark. For the strongest contenders, we provide a more detailed analysis under both lenient and strenuous conditions. All tests were performed on an Nvidia GTX Titan X (Pascal). Additional test results for Nvidia Maxwell and Kepler architectures can be found in Appendix A.

4.6.1. Initial Runtime Comparison

Our initial microbenchmark performs 10 alternating enqueue-dequeue pairs over a varying number of concurrently running threads. Due to this ideally balanced setup, we can include blocking queues into the test, as neither **Empty** or **Full** states are reached. Figure 4.2a shows the average achieved runtimes. Due to their high register usage, LCRQ and WFQ reach the maximum number of concurrently running threads at 43 008 threads on the Titan X (Pascal), *i.e.*, they achieve 37% less occupancy than the other approaches. Since SFQ can only execute at a per-warp granularity, we repeat the above experiment with only one thread in each warp accessing the queue (Figure 4.2b).

These initial experiments confirm that non-blocking strategies, based around the concept of optimistic concurrency control, do not work well with thousands of threads. All four non-blocking queues built around optimistic C&S (TZQ, SHCQ, NSFQ, and MSQ) are trailing significantly behind the others. Even a queue that allows only two threads concurrent access (2MQ) can be significantly faster. However, as the number of concurrent threads approaches maximum occupancy, all of the above techniques are more than 1000× slower than the remaining algorithms with per-thread granularity, and more than



(a) Runtime of all queues, with thread granularity

(b) Runtime of all queues, with warp granularity

(c) Details for fastest queues, with thread granularity

(d) Details for fastest queues, with warp granularity

Figure 4.2.: Runtime performance results of all queues for 10 enqueue/dequeue operations.

100× slower with per-warp queuing interaction. Clearly the fastest runtimes for these initial test are obtained by our queues, as well as GQ and the recently proposed SFQ, LCRQ and WFQ. Note that we have omitted BWD and BSQ from these plots, since they exhibit virtually identical behavior to BQ in this balanced scenario.

A closer look at the runtime performance of the faster contenders is given in Figure 4.2c and 4.2d, which include lowest and highest measured runtimes as overlay. Although GQ is non-linearizable, it trails behind BQ with a slowdown of more than 2× for launch configurations exceeding 45 056 threads (or 1504 warps), caused by continuous modification of the two counters in addition to front and back pointers. LCRQ and WFQ have a higher base cost than all other techniques (3–6× compared to BQ) and quickly deteriorate at per-thread granularity, but catch up with the non-linearizable GQ per-warp. With an increasing number of threads accessing the queue, LCRQ also shows the highest variance in runtime. Per-warp, LCRQ continuously loses its advantage over WFQ's higher base cost. Although SFQ is conceptually much simpler and less versatile than our queue, it is still narrowly outperformed by the BQ. For launch configurations with > 512 warps, we found a relative slowdown between 1.4% and 9%. We ascribe this fact to BQ not having to poll a closed state, unlike SFQ. Overall, BQ poses the fastest queue in this scenario.

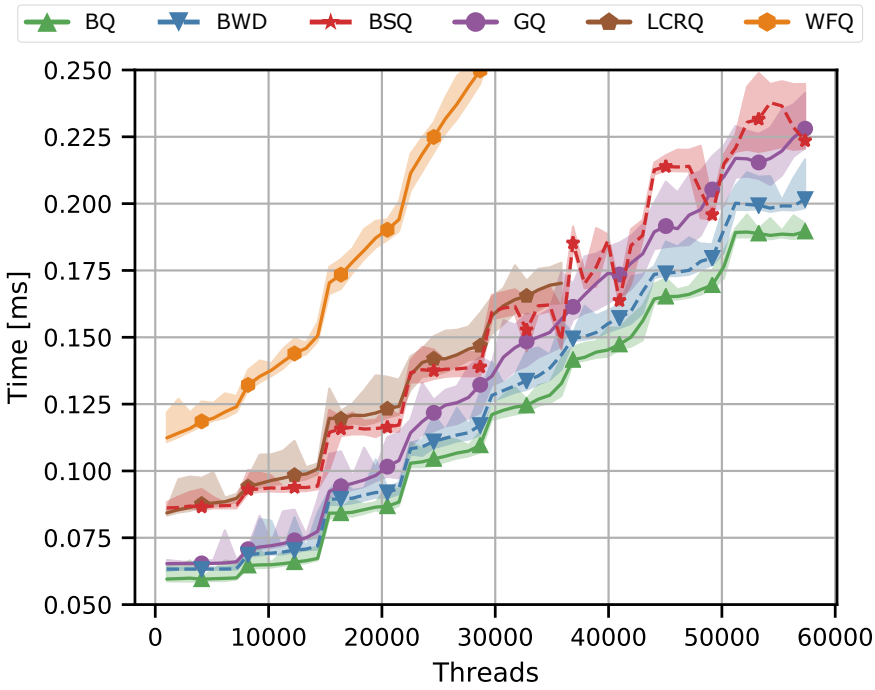
4.6.2. Imbalanced and Real-world Scenarios

In order to prove their usefulness in a realistic work scheduling scenario, queuing algorithms must be able to reliably handle cases where each task produces a certain amount of work, the number of enqueues and dequeues is not balanced, and queues can actually run empty.

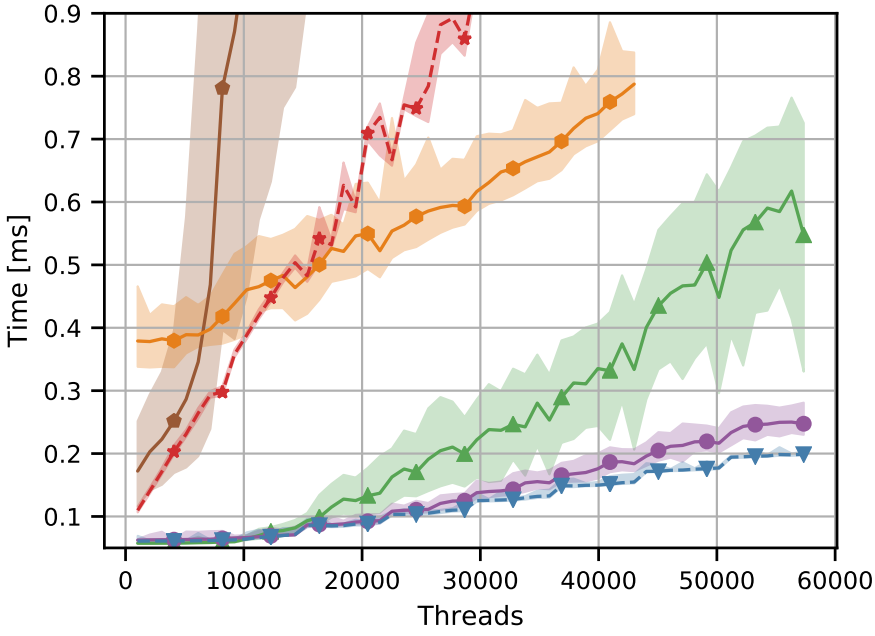
Synthetic Benchmark To produce imbalanced scenarios, we first extend our initial test case such that every thread randomly performs between 1 and 10 loop iterations where `enqueue` and `dequeue` themselves are called with reduced probability. Consequently, the number of enqueue and dequeue operations is no longer balanced, which introduces the possibility of underflow. Furthermore, we simulate a workload for each task by executing 128 fused multiply-add (FMA) instructions after each successful `dequeue`.

Since this scenario requires threads to recover from underflow in order to finish the test, they cannot be evaluated for the blocking SFQ. Also note that we avoid overflow in this test by allocating sufficient memory for all queues. LCRQ, which is based on maintaining smaller linked buffers, reacts to an overfull buffer by allocating and initializing new ones. In order to mask this dependency on dynamic memory, we pre-allocate and initialize a chunk 32× the size of the other queues to provide LCRQ with enough resources.

We show our results for imbalanced test cases in Figure 4.3. With simulated workload added, the differences across techniques diminish for the default behavior. This can be observed in Figure 4.3a, where a 2× higher probability of `enqueue` than `dequeue` ensures that every thread can perform its task without delay, given that overflow does not occur. In the opposite case—probability of `dequeue` exceeds that of `enqueue` 2×—underflow occurs, and performance figures change considerably (Figure 4.3b). Since there is never a substantial amount of work to steal, BSQ keeps unsuccessfully checking other queues, and its overhead is never amortized. The non-blocking techniques LCRQ and WFQ are at least 2.5× slower than all other techniques. Note that both approaches behave destructively, as queue slots can become unusable if a `dequeue` arrives there before an `enqueue`. In contrast to LCRQ, WFQ counteracts slot thrashing with its slow-path/fast-path dynamic, by turning unsuccessful `dequeue` threads into `enqueue` helpers. This is reflected by its runtime rising ∼6× slower than LCRQ at underflow. However, BQ significantly outperforms both approaches and hence poses the fastest linearizable queue of those tested.



(a) $\mathbb{P}(enq) = 50\%, \mathbb{P}(deq) = 25\%$



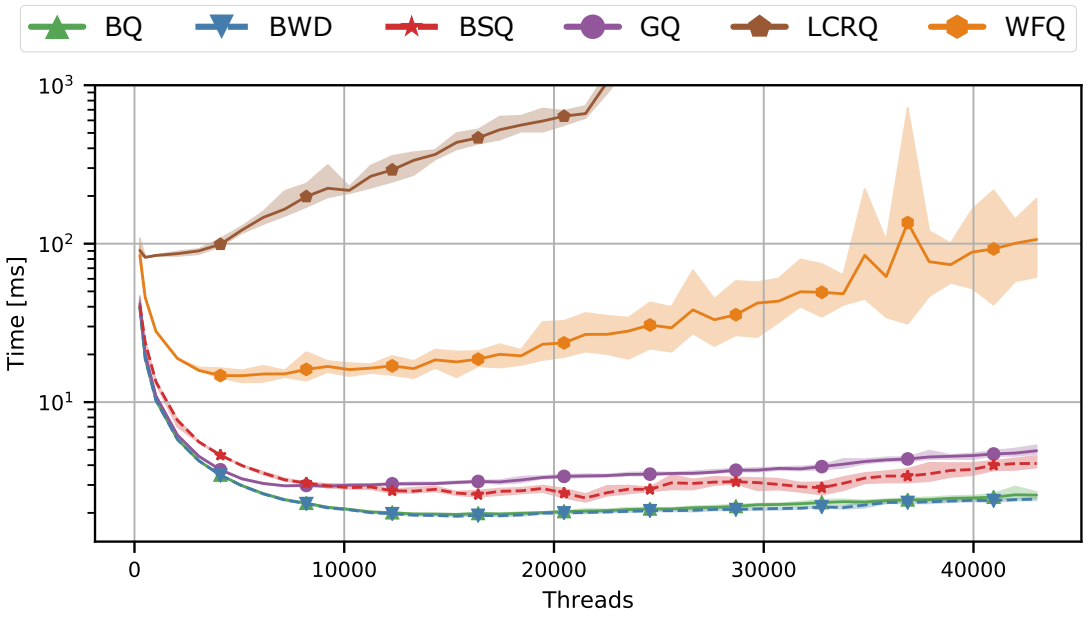
(b) $\mathbb{P}(enq) = 25\%, \mathbb{P}(deq) = 50\%$

Figure 4.3.: We consider imbalanced test cases, both synthetic and realistic. We test enqueue with probability $\mathbb{P}(enq)$ and dequeue with $\mathbb{P}(deq)$ on initially empty queues. In (a), $\mathbb{P}(enq) > \mathbb{P}(deq)$ and the queues never run empty. Using a constant workload reduces the performance gap compared to the initial benchmark. In (b), queues quickly hit underflow, which has a devastating effect on the performance of LCRQ and WFQ, and, to a much lesser extent, on BQ.

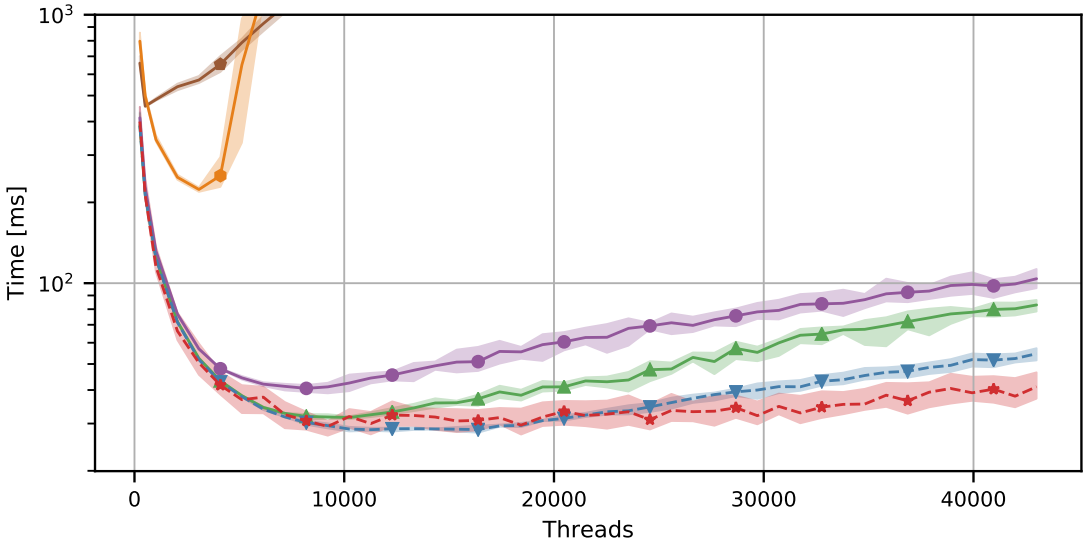
The simpler, non-linearizable GQ achieves up to 64% faster runtimes than BQ, but is also prone to erroneously detecting empty states. In a real-world scheduling scenario—where the workload is not known beforehand—this may cause threads to quit prematurely, compromising performance and correctness. The fastest runtimes are reported for our non-linearizable version of BQ, the BWD (10% faster than GQ at maximum occupancy). Compared to GQ, underflow detection by the BWD is less problematic, since it makes all interactions immediately visible via the *Count* variable. Hence, it follows that the BQ/BWD pair provides the best choice for a linearizable/non-linearizable queue, respectively, in imbalanced scenarios.

Page Rank In order to provide a real-world example, we evaluate all competitive queues that are capable of handling underflow on the computation of page rank for two directed networks. Specifically, we compute the first 8 iterations for the data sets *p2p-gnutella31* and *web-NotreDame*, provided by the Stanford Large Network Dataset Collection (Leskovec and Krevl, 2014). The development of queue performance with increasing thread count is plotted in Figure 4.4. We pre-fill queues with one work item per node and launch a persistent kernel that tries to dequeue elements, until all threads agree that no more work is being generated. Active nodes pass on their latest available page rank value to their neighbors. If a node N finds that it is the last to contribute to the page rank of another node M in iteration i, M is enqueued for iteration $i+1$. In order for LCRQ and WFQ to run fairly stable without immediately consuming all available memory, we had to shrink their buffer segments below the suggested size ($<$128 slots per segment). We also added traversal of previous *Head* pointers to LCRQ to reclaim abandoned segments.

Performance measurements for tested queues are shown in Figures 4.4a and 4.4b. LCRQ/WFQ quickly fall behind, with slowdown of at least $60/6\times$ over GQ, BQ and its variants in *p2p-gnutella31*, and $95/400\times$ in *web-NotreDame* for more than 10 240 threads. In both networks, BQ outperforms GQ. This confirms our assumption that GQ's erroneous underflow detection is detrimental for tasks that only terminate when no more data is produced, which holds for the Page Rank test (in contrast to our synthetic tests). Consequently, BQ and BWD are consistently 10–15% faster than GQ for configurations $>$10 240 threads. Furthermore, we find that BSQ performs best for the large *web-NotreDame* network at maximum occupancy (9% over BWD). This is due to new work being generated in bursts when a neighborhood of nodes finish an iteration simultaneously, allowing for work stealing to take effect.



(a) Page Rank for *p2p-Gnutella31*



(b) Page Rank for *web-NotreDame*

Figure 4.4.: Testing queues in a real-world example to compute 8 iterations of page rank, we find that our algorithms (BQ, BWD and BSQ) are the fastest available techniques. For both medium-sized networks (a, 60k nodes) and large ones (b, 300k nodes), our queues achieve lower runtimes than simpler alternatives (GQ).

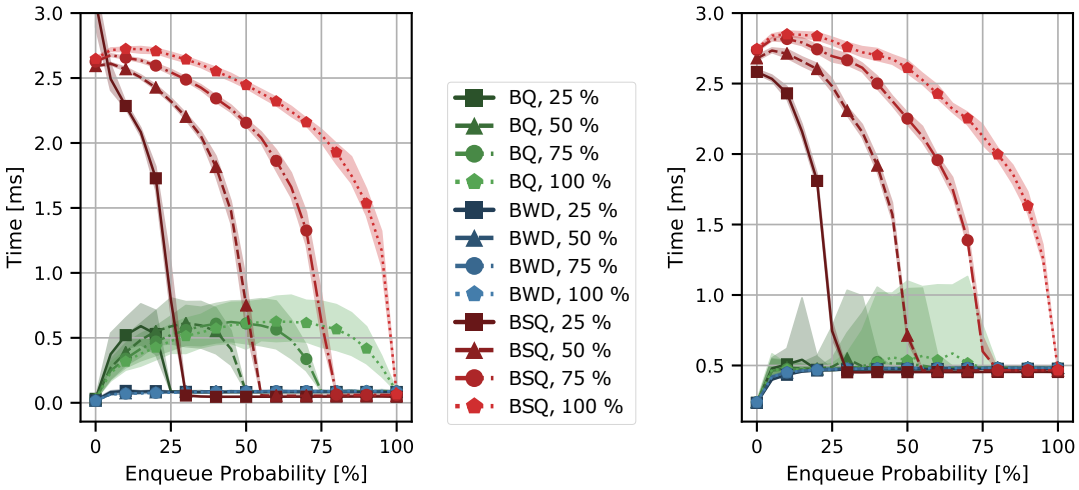
4.6.3. Broker Queue Variants Comparison

To investigate differences in behavior between BQ, BWD and BSQ in detail, we test various enqueue and dequeue probabilities under maximum occupancy. Figure 4.5a shows that, if `enqueue` probability is higher than `dequeue`, there is negligible difference in queue performance among the three approaches (<10%, thus within usual variance), with BSQ being marginally faster due to reduced contentions. However, at lower `enqueue` rates, the performance of BSQ suffers considerably (up to 30× slowdown). This is explained by its modus operandi: at maximum occupancy, a high number of thread blocks (and thus distributed queues) is employed. Hence, with few work items being generated at all times, work stealing constantly checks many queues, just to determine that they are all empty.

The largest difference between the BQ and BWD queues can be observed when `dequeue` happens about twice as often as `enqueue`. At this point, every other `dequeue` attempt observes a potential **Empty** state (BQ up to 5× slower). It is unlikely to observe an actual underflow of the queue, as there are still many `enqueue` operations happening, leading to multiple fail-and-retry attempts. For lower `enqueue` probabilities, it is easier to observe **Empty** and thus the performance of BQ normalizes. Similarly, for higher `enqueue` probabilities, it is also more likely for a `dequeue` to immediately succeed. It follows that ensuring linearizability of BQ can increase runtime by up to 20× if the queue is nearly empty/full all the time. However, already a small simulated workload (320 FMA operations) reverses this trend (Figure 4.5b): under load, BSQ shows lower relative slowdown (∼6×), and **Empty**/**Full** are likely matched by the pointers, as fewer threads access the queue concurrently. Hence, average performance of BQ and BWD is nearly identical.

4.7. Discussion

In this chapter, we presented queuing strategies geared towards effective work distribution on the GPU: the broker queue, as well as two simpler, optimized variants. Previous work in this field usually follows one of two strategies: relying on optimistic concurrency control and thus being non-blocking, or showing strict blocking behavior, even when the queues are full or empty. While the former shows poor scalability in massively parallel environments with thousands of threads, the latter prohibits effective scheduling mechanisms for work distribution on the GPU.



(a) Runtimes with no workload

(b) Runtimes with 320 FMAs workload

Figure 4.5.: Performance comparison of BQ, BWD and BSQ at different enqueue/dequeue ratios shows that BSQ in general reduces contention in ideal cases, but suffers from massive overhead otherwise. (a) As BQ is hitting a potentially empty queue, it waits until the state is verified, which reduces its performance, especially if that state is likely to change again. (b) This effect becomes smaller as the task workload (simulated by 320 FMAs after each dequeue operation) increases.

Instead of following either strategy, we have combined the most desirable features of both, keeping the scalability of blocking queues, while ensuring versatility through non-blocking detection of under- or overflow. Comparing to an extensive body of previous work, we found that our techniques consistently rank among the most competitive approaches. Since the broker queue was conceived with GPU hardware in mind, it does not rely on exotic or impractical hardware features, rendering its implementation straightforward. Our evaluation showed the broker queue to be the fastest linearizable queue for distributing work on the GPU in various scenarios. In balanced and realistic setups, the broker queue outperformed all previous algorithms. We also presented an even faster, non-linearizable variant of the broker queue, for the purpose of general work distribution: the broker work distributor. In terms of performance, the broker work distributor surpassed all previous approaches, even in synthetic imbalanced scenarios. Adding work stealing on top of our queue can ideally increase efficiency even further under realistic load. Although our proposed algorithms do not fulfill the non-blocking property, they are resilient to under- and overflow scenarios, making them prime candidates for work distribution and dynamic load balancing on the GPU. In fact, the queues described in this chapter can be effectively applied in any scenario where a fast, concurrent queue is needed.

5. Hierarchical Bucket Queuing and Adaptive Rendering on the GPU

Contents

Armed with a queuing algorithm that delivers high-end performance in a variety of scenarios, we seek to apply our solution in a parallel scheduling framework. Our goal is to leverage fine-granular work prioritization fit for adaptive rendering applications while minimizing runtime overhead.

5.1. Adaptive Rendering & Priority Scheduling

The static nature of GPU execution is a severe limitation, even in its core application of interactive graphics. For example, virtual reality applications demand low-latency, real-time rendering for high-resolution, head-mounted displays (HMD). Long latencies or spikes in rendering time can lead to severe discomfort. According to current beliefs (Hunt, 2015), foveated rendering—adaptively rendering regions around the user's fixation point in higher resolution—is one way to address these issues. However, using a traditional rendering API, one can only rely on predictions about how long rendering might take to decide upfront on an allocation of computing power that might meet a given deadline, without guarantee. One solution could be a progressive renderer with dynamic priorities. It could adaptively focus computing power around areas of interest. Rendering can stop, as the deadline of acceptable latency is reached, maximizing the quality achieved within a given time frame. This idea is not new. Prioritized rendering has been investigated in previous work, *e.g.*, for the goal of reducing sampling rates in path tracing (Mitchell, 1987; Overbeck, Donner and Ramamoorthi, 2009; Rousselle, Knaus and Zwicker, 2011). However, most of these approaches were conceived with no or at best low degrees of parallelism considered. The peculiarities of the GPU require that load balancing and prioritization be considered in a whole new light. Scheduling a task on the GPU means mapping it to blocks or warps. A large number of tasks is typically required to make efficient use of the massively parallel GPU. The execution time of each task should be kept short to avoid branch divergence, hence we cannot rely on slow, exact solutions for prioritization (*e.g.*, Softshell) and hope to hide sorting latency with excessively long tasks. Coarser prioritization (*e.g.*, Whippletree) usually fails to capture important differences in priority for individual work packages.

The sum of these facts makes prioritized load balancing an interesting but challenging problem for deployment on the GPU: There is usually a large number of short tasks to be managed, thus, scheduling decisions must be made very often. Making scheduling decisions involves reading information about the execution state from memory, which is very costly. Due to the massive parallelism and short task durations, tasks enter and leave the scheduling system in parallel at a high rate. Constant reorganization of shared data structures such as sorted queues or heaps is problematic, as locking must be avoided. On the GPU, a priority queue in the original sense is not even a theoretical possibility, as there is no absolute order that could be established for thousands of parallel operations carried out at any given point in time.

Thus, GPU priority scheduling can only aim to execute tasks with higher priority before tasks with lower priority *on average*. Hence, the effect of a fine-granular, approximate ordering of tasks according to their importance may be virtually indistinguishable from exact sorting in a massively parallel environment. However, even an approximate priority scheduling solution can enable a variety of applications to benefit from adaptive behavior. While following such a relaxed prioritization policy can be expected to simplify the task of designing an analogous implementation in a parallel environment, we still need to consider and overcome the aforementioned issues. We tackle them with our approach based on a hierarchical organization of bucket queues and make the following contributions:

- We present a flexible, hierarchical queuing structure for the GPU that can be configured to implement a variety of scheduling policies and is efficient for massively parallel access.
- We expose our bucket queues through a scheduling control model consisting of three simple entry points that allow for an easy and efficient implementation of different scheduling policies. Our model can be plugged into any persistent threads solution and would also lend itself to implementation in future hardware.
- We investigate different methods to implement fair-scheduling, earliest-deadline-first scheduling, and user-defined scheduling policies on top of our approach and compare the performance of each policy in a set of synthetic tests with previous work.
- We show how priority scheduling can be used for elaborate load balancing with guaranteed latency for foveated micropolygon rendering as well as adaptive path tracing.

The exposition in this chapter follows three logical steps which define the sequence of the following sections. First, we will describe the individual methods required for hierarchical bucket queuing, along with our frame of reference for the definition of priorities and the programming interface to our implementation in a parallel C++/CUDA framework. Next, we will assess the general efficacy of our relaxed prioritization scheme for realizing basic task scheduling policies on the GPU in comparison with previous work. We discuss trends, strengths and weaknesses, as well as beneficial conditions and configurations for our work, relevant alternatives and considerations for a theoretical implementation in dedicated hardware. Finally, we will show how our solution can be used to introduce adaptive behavior to GPU software rendering applications while ensuring dynamically balanced workload.

5.2. Hierarchical Buckets for GPU Scheduling

Work queues are widely accepted as the standard tool for load balancing on the GPU. They are used by software schedulers and the GPU hardware, as shown in Figure 5.1. They can be filled by the CPU and also directly by kernels executing on the GPU (Jones, 2012). Task-specific work queues are further useful, as they allow for work aggregation (Steinberger, Kenzel, Boechat et al., 2014). The goal of this chapter is to design a priority scheduling scheme that integrates and enhances these existing practices. We identify the following set of requirements a queue-based system must fulfill to achieve this goal:

R1 While one application might require a single queue and permit mixing different tasks, another application might need multiple queues to aggregate tasks of different types. *Thus, priority scheduling mechanisms and the queuing back-end must be customizable.*

R2 When a multiprocessor requests work, the GPU-wide scheduling must not block, as this would halt other multiprocessors concurrently requesting work. *Consequently, a separate, serial priority scheduler between queues and multiprocessors cannot be used.*

R3 Current strategies for GPU scheduling allow tasks to be generated and inserted into the queues at any point by any thread on the GPU. To avoid stalls, producers must be able to add work in parallel without interference. *Hence, complex data structures that require locking whenever a new task is generated are not an option.*

R4 Peak performance can only be achieved if a multiprocessor does not stop executing tasks unless all tasks have been processed. *Thus, execution must not be halted for priority scheduling*, i.e., *it is not possible to use periodic rebuilds of data structures or very complex scheduling algorithms.*

5.2.1. Hierarchical Buckets

We propose the use of multiple instances of an efficient queue, such as the broker work distributor, organized into a hierarchical structure of buckets. By making the hierarchy configurable by the application, it is possible to cover a variety of scenarios. Every node of the constructed hierarchy can have an arbitrary number of children, *i.e.*, it is possible to place any number of buckets within another bucket. Leaf nodes in the resulting tree, *i.e.*, the terminal buckets, correspond to queues storing the actual tasks. Buckets can

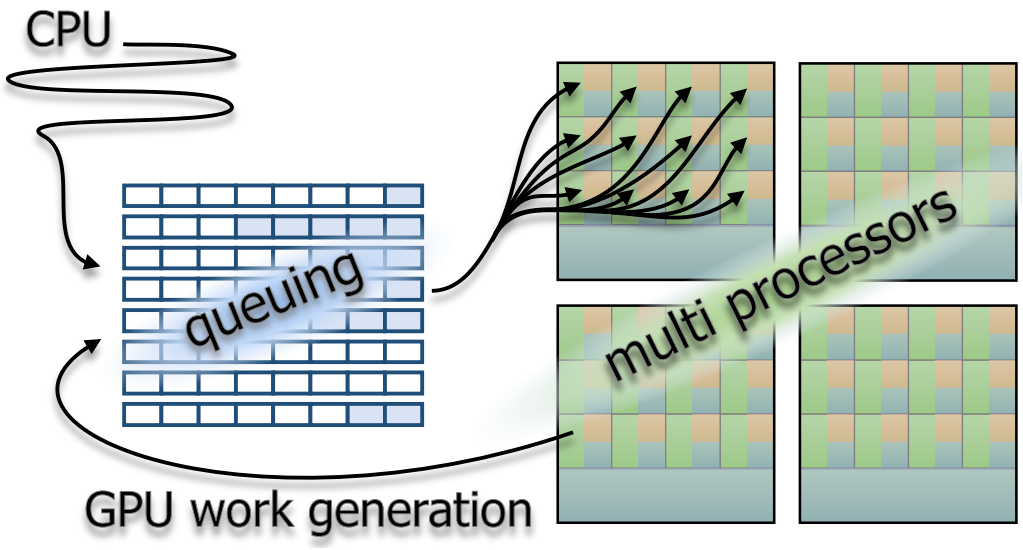


Figure 5.1.: To keep the multiprocessors of the GPU busy, work is usually managed in queues located on the GPU. These queues can be filled by the CPU as well as the GPU.

be limited to certain types of tasks, allowing each queue to be optimized for the subset of data types it has to hold. Adding an element to a bucket queue hierarchy corresponds to walking down the tree until a leaf node is found and pushing into the corresponding queue. To demonstrate the universality of such a configurable system, we have provided several examples in Figure 5.2: (a) The simplest configuration corresponds to a single bucket containing all types of tasks, which is analogous to previous-generation GPU command queues and simple persistent threads approaches, as well as Softshell (Steinberger, Kainz et al., 2012). (b) A group of multiple child buckets all taking the same tasks corresponds to current-generation GPU work queues. (c) A list where buckets accept one specific task type (*i.e.*, they perform the same instructions) maps to the behavior in Whippletree (Steinberger, Kenzel, Boechat et al., 2014). (d) By using a two-level hierarchy, the first level can implement a discrete set of priorities, while the second level can collect tasks of different types. This way, it is possible to combine priority scheduling and work aggregation for load balancing. (e) In a more complex setup, tasks for two different processes can be stored (P_0 and P_1). While P_0 enqueues all tasks indiscriminately, P_1 supports organizing tasks with different priorities. Note that in this example, certain priorities are only assumed by particular tasks (distinguished by color). At the lowest level, the queues can distinguish between small tasks (S) that need to be aggregated to fully utilize a multiprocessor and larger tasks that are stored in a combined queue (C) with other task types.

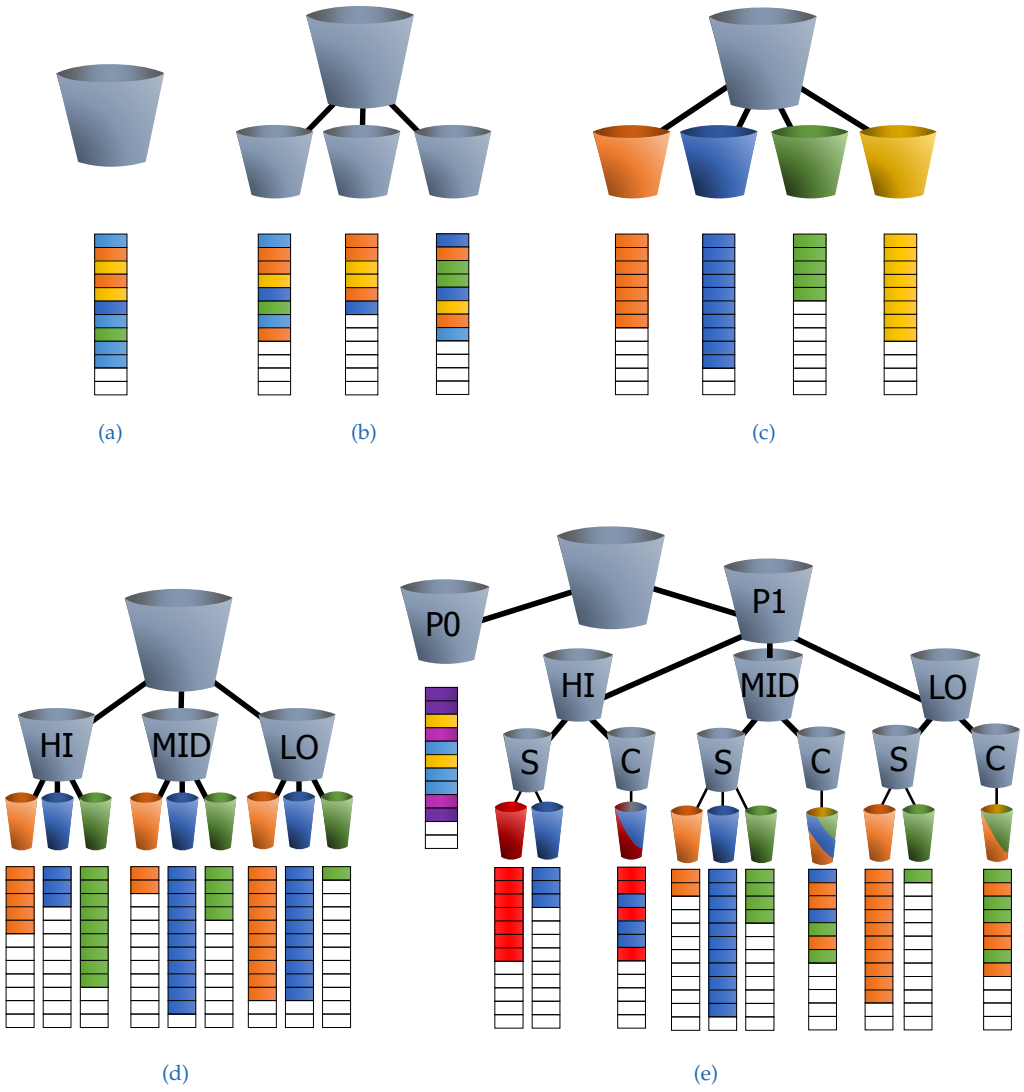


Figure 5.2.: Hierarchical bucket queues can (a, b) capture simple setups similar to the work-dispatching mechanisms found on the GPU, (c) collect tasks according to their type, (d) combine prioritization with work aggregation, and (e) implement sophisticated structures. Note that colored terminal buckets can be specifically optimized for the subset of data types they are supposed to hold.

5.2.2. Customizable Priorities

While the fundamental concept of a queuing hierarchy enables a variety of applications in itself, we need to define an efficient and easy-to-use scheme for establishing priority-based load balancing on top of it. Taking into account **R2-R4**, we propose a configurable, lightweight priority scheduling model based on the observation that a persistent threads megakernel fed from a queueing hierarchy enables the following three forms of scheduling:

1. When a new task is generated, the bucket hierarchy is traversed and the appropriate queue to store this task can be selected.
2. When a multiprocessor finishes a task (or a set of tasks), the bucket from which to retrieve the next task can be selected.
3. During execution, a small number of maintainer threads can update the queues by reorganizing elements in the background.

Given these possibilities, a variety of sophisticated dynamic load balancing strategies can be realized. Being able to choose one of several queues during enqueue allows for (discrete) sorting according to arbitrary criteria. Being able to choose which queue to dequeue from allows the system to make decisions based on the current execution state, which might have changed since the tasks were enqueued. For example, tasks could be sorted into queues according to which process they belong to when they are generated. As a working block dequeues the next task, it could choose according to how much processing power each process has consumed in the meanwhile to, *e.g.*, achieve fair scheduling. In addition to these fundamental control mechanisms, dedicated maintainer threads can be deployed to continuously adjust the order of tasks within queues according to a constantly changing metric.

With our approach, we imagine all three forms of scheduling to be freely programmable, similar to the way shaders bring programmability to a graphics pipeline: before execution, task types are defined, the bucket queue hierarchy is set up and user-defined callback functions that implement each type of scheduling decision are registered. These functions are called whenever (1) a new task is generated, (2) a multiprocessor requests new data, or (3) during queue maintenance. Our current implementation employs a persistent threads megakernel approach similar to Whippletree (Steinberger, Kenzel, Boechat et al., 2014), using their basic framework with our own scheduling mechanisms added in. We also use their basic queue implementation, the broker work distributor, in the leaf nodes of the bucket hierarchy. Note that any other persistent threads approach and queue could be used here instead.

5.2.3. Enqueue

Whenever a new task is generated, we want to enqueue it efficiently, while still allowing for control by the application. Given that the bucket hierarchy is known at compile-time, enqueue can be completed with a single traversal of the hierarchy. Callbacks registered with every bucket in the hierarchy are called during traversal of the tree to decide which child bucket to choose next. When a leaf node is reached, we enqueue the task into its associated queue.

Consider the example in Figure 5.2e: The callback for the first bucket returns which process the task belongs to. In case the task comes from P1, the next callback determines if it is of high, medium, or low priority, before deciding if the task is large enough to be stored in a combined queue or if it should be merged with others of the same kind. Note that we could also only provide a single callback determining the final queue. However, enforcing a hierarchy makes it easy to reuse and extend an existing scheduling policy. For example, if another process with a different set of queues is added, it will be represented by a new branch under the first bucket, and no changes to the remaining parts of the hierarchy are needed.

As long as the underlying queue implementation supports concurrent enqueue operations, our priority scheme fulfills **R3**. In case that there is no space available in a found terminal queue, there are different possibilities to recover. We can walk back up the hierarchy while executing the callbacks with a reduced set of choices, return that the enqueue failed due to lack of free storage, or keep retrying the enqueue operation until a slot is available. Since the first option is difficult to implement for the user and the third option bears the risk of deadlocking, we opt for the second option and consider the case of a failed enqueue an exception that the application has to handle.

Tasks can also be generated from the CPU, *e.g.*, via a traditional kernel launch, or an initial set of tasks that set in motion a more complex dynamic algorithm. In this case, we can either traverse the bucket hierarchy on the CPU and transfer the tasks to the appropriate queues on the GPU, or launch a kernel where each thread is responsible for generating and enqueuing one or more tasks. While the first option can generate tasks even while the persistent megakernel is running (L. Chen et al., 2010), the second option enables handling of multiple tasks in parallel. As the first option usually involves a performance overhead and the Whippletree programming model only supports the second, we also limit our implementation to that case.

To optimize the traversal process, we skip the evaluation of any callbacks if there is only a single viable choice, *e.g.*, a multi-bucket setup where each bucket is constrained to a specific task type (compare Figure 5.2c). Even if callbacks are evaluated, their execution is usually very efficient, since the task's payload will normally have already been moved into local memory and no additional transactions are required. If the underlying system were to be implemented by a hardware scheduler, it would still make sense for the callbacks to be evaluated on the compute cores of the GPU. As task generation (or kernel launches via dynamic parallelism) happens during kernel execution, the thread generating the task can also immediately perform the traversal.

5.2.4. Dequeue

Similar to enqueue, we expect callbacks for dequeue to be registered with every bucket. When a multiprocessor finishes its previous work and requires new tasks to be dequeued, a possible solution would be to traverse the hierarchy top-down. However, empty terminal queues are expected to be a common case with this strategy, especially with a large number of buckets, prolonging the search for available tasks. Furthermore, due to the SIMD nature of execution on the GPU, there is always at least a full warp of threads available when fetching new tasks. Hence, we can make use of these otherwise idle threads to implement a parallel bottom-up traversal for dequeue: every thread starts at a different leaf and walks up the hierarchy. At each bucket, the information coming from each of its children is combined by the respective callback until the final load balancing decision is computed at the root. This whole process basically corresponds to a parallel reduction, enabling us to find the most relevant non-empty queue. Once the queue to dequeue from has been chosen, we compute the number of tasks to be fetched such that the currently available threads on the multiprocessor all receive sufficient work. Next, we try to dequeue the respective number of tasks from the queue. However, other worker blocks may have consumed all tasks from that queue during the time spent in traversal. In this case, we restart the dequeue process and mark the now empty queue. We found this trial-and-error approach to be much more efficient than locking queues during traversal (**R2**).

The bottom-up approach is not only more efficient when dealing with a large number of queues, but turns out to be a good approach for implementing many more complex load balancing strategies in general. Consider two single-threaded task types, each having its own bucket—one high priority, the other

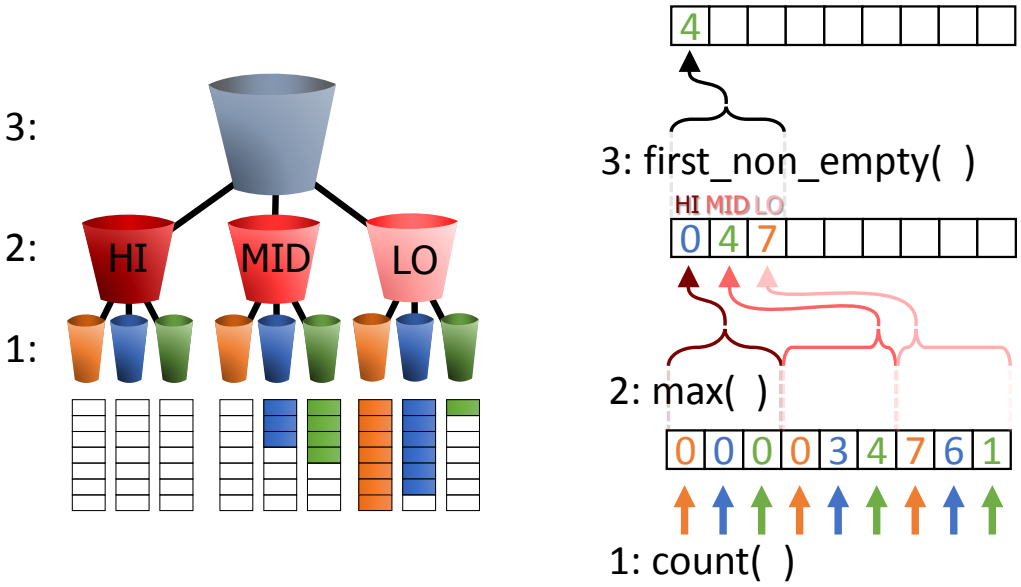


Figure 5.3.: Dequeue example: (1) the callback for the leaf buckets returns the number of elements in the queue; (2) the callback for each prioritized bucket on the middle layer selects the child queue with the highest element count; (3) the root bucket selects the child with the highest priority reporting a non-empty queue. In this example, dequeue would choose the green bucket with MID priority.

low. Assume that there is one task in the high priority queue and several in the low priority queue. A sophisticated scheduler might want to choose to execute multiple low priority tasks instead of a single high priority task, as it can make better use of the available processors. If we used a naïve top-down traversal, it would be difficult to implement such a behavior across multiple hierarchy levels, as this would require descending into every leaf and keeping track of all the results computed along the way. With the bottom-up approach, however, such behavior can be achieved in a very simple and natural way. The full process in an exemplary setup is illustrated in Figure 5.3.

If this bottom-up scheme were to be implemented in hardware, we would also consider performing the traversal on the compute units of the GPU, since that would make the full instruction set available to the callbacks. Considering that some warps usually finish before others, there is potential to hide the added latency from the evaluation of the callbacks, by having the first warp to complete the previous task immediately start evaluating the dequeue callbacks. This is likely to lead to new tasks already being available as soon as the remaining warps complete.

5.2.5. Maintain

While a large number of important load balancing strategies can be implemented by enqueuing and dequeuing logic alone, there are cases that require changing the order of elements already in queues, *i.e.*, sorting the queues. While fully sorting the queues is not a viable option (*cf.* **R4**), Steinberger, Kainz et al. (2012) showed that progressive sorting can be used to gradually rearrange queue contents in a non-blocking fashion. We adopt this approach, allowing each queue to be flagged for automatic maintenance by registering a callback for it. Given an item that is currently stored in the queue, the callback must return a numerical priority value that can be used for sorting.

If any queue is marked for maintenance, we dedicate a configurable number of GPU threads to continuous sorting. Instead of sorting the entire queue at once, only tasks within a limited sorting window are considered at each point. A sorting pass progressively advances the sorting window from the back of the queue to the front, as proposed in the original approach (Steinberger, Kainz et al., 2012) and outlined in Figure 5.4. Once a sorting pass is finished, sorting is restarted at the back of the queue. To avoid stalling dequeue operations, we uphold a safety margin to the very front of the queue. Multiple queues are handled by simply cycling through them. Some queues, however, might require more attention than others. Therefore, we record the actual number of exchange operations carried out during one sorting pass and the number of new elements received since last restarting the sort. Using these values, we prioritize the sorting of those queues which we expect to contain the highest number of unsorted elements.

In practice, we reserve a single GPU worker block to be used as a dedicated maintainer. For an integration with a hardware scheduler, a programmable unit would be required if the callback needs to be reevaluated during each sort, *i.e.*, if priorities of queued tasks are allowed to change. In this case, we would suggest assigning a small block of threads to maintenance, similar to the current software approach. However, if priorities are static, dedicated maintainer threads can be avoided. Instead, the priority of each task can be computed when it is enqueued and stored alongside the task. The priorities can then be read from memory and sorting can be implemented by a hardwired unit instead.

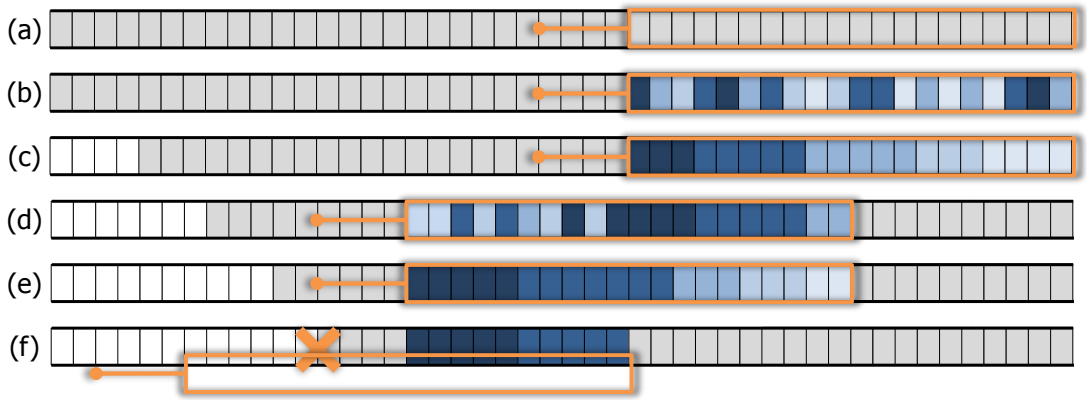


Figure 5.4.: Progressive sorting of a queue using a sorting window (orange) with safety margin to the front, while tasks are concurrently removed from the queue; time steps (a)-(f): (a) for all tasks in the sorting window, the callback delivers priorities (dark blue to white); (b) the returned values are sorted locally; (c) tasks are exchanged according to the local sorting; (d) the sorting window is advanced to the front; (e) the next segment is sorted; (f) the safety margin reaches the front of the queue, sorting cannot continue without stalling the execution and must be canceled and restarted at the back.

5.2.6. Application Programming Interface

Since we use Whippletree's execution model, our API builds upon its template-based CUDA/C++ interface. In Whippletree, a task can be defined as follows (slightly simplified):

```
1 struct Task {
2   static const int NumThreads;
3   typedef int Payload;
4   __device__ static
5   void execute(int tid, Payload& data);
6 };
```

When chosen for execution, the task's `execute` method is called by the requested number of threads, all receiving the dequeued payload as input.

Following the spirit of a C++ interface, the bucket hierarchy and callback functions are also set up using templates in our API. Both buckets (`Bucket`) and queues (`Queue`) expect a template class that specifies the callback functions.

```
template<class LeafCB, class... Tasks>
struct Queue;

template<class BucketCB, class.. Children>
struct Bucket;
```

For leaf nodes, two callbacks can be provided:

```
struct LeafCallback {
  template<class Queue>
  __device__ static
  CustomType checkLeaf(const Queue& q);

  template<class Task>
  __device__ static
  Comparable maintain(Task::Payload& data);
};
```

The `checkLeaf` callback is called during dequeue and provides the information that is propagated up the hierarchy. The `maintain` callback is optional. Only if it is present will the maintainer attempt to sort the queue. Note that the return value of this method can be chosen freely, the only requirement is that a suitable comparison operator exists.

The bucket's `traverse` callback for enqueuing and its `propagate` callback for dequeuing have the following signature:

```
struct BucketCallback {
  template<class Task>
  int traverse(Task::Payload& data);

  int propagate(CustomType* infos);
};
```

Note that `traverse` itself is a template and can be specialized for different tasks. Also note that `CustomType` can be of any type as long as it is compatible with the return type of `checkLeaf`.

Once a user has set up the callback functions, the queuing hierarchy can be established. The following example shows how Figure 5.2d can be defined with just a few lines of code, using the callback definitions `LeafHasData`, `RoundRobin`, and `Discrete`, which we discuss in the upcoming section:

```
typedef Queue<LeafHasData, Task1> Queue1;
typedef Queue<LeafHasData, Task2> Queue2;
typedef Queue<LeafHasData, Task3> Queue3;
typedef Bucket<RoundRobin<3, AlwaysFirst>,
  Queue1, Queue2, Queue3> B3;
Bucket<Discrete<3>, B3, B3, B3> Root;
```

Note that each terminal queue only stores the payload for a single task type. `B3` defines a bucket that has three queues as children. By adding `B3` three times to the root node, three instances of `B3` are created as immediate child buckets of the root.

Given the bucket hierarchy root node, our implementation generates the load balancing logic from the user-provided callbacks and combines it with the task `execute` functions into a megakernel. If at least one maintainer callback is provided, additional routines are added to the megakernel for turning the block with id '0' into the maintainer upon kernel launch.

5.3. Scheduling Policies

We now show how the discussed queuing framework can be brought to use in practice. First, we demonstrate how simple scheduling mechanisms, *e.g.*, discretized prioritization and round-robin can be set up with bucket queues. After that, we focus on more advanced examples such as fair scheduling, earliest-deadline-first, and user-defined policies. We also compare our results to previous work. For our evaluation, we used an Intel i7-4771 with 16GB RAM running Windows 10 and an Nvidia Geforce GTX 980Ti.

5.3.1. Discretized Priorities

A simple way to build priority scheduling is using a fixed number of buckets, each for a different priority. Whenever a new task is created, its priority value is computed and it is inserted into the appropriate bucket. During dequeuing, available buckets are probed in descending order of priority. Assuming priorities, *e.g.*, in the range $[0,1)$, we discretize them linearly according to the total number of available buckets. See Figure 5.5 for an example using four buckets, each covering one quarter of the total priority range.

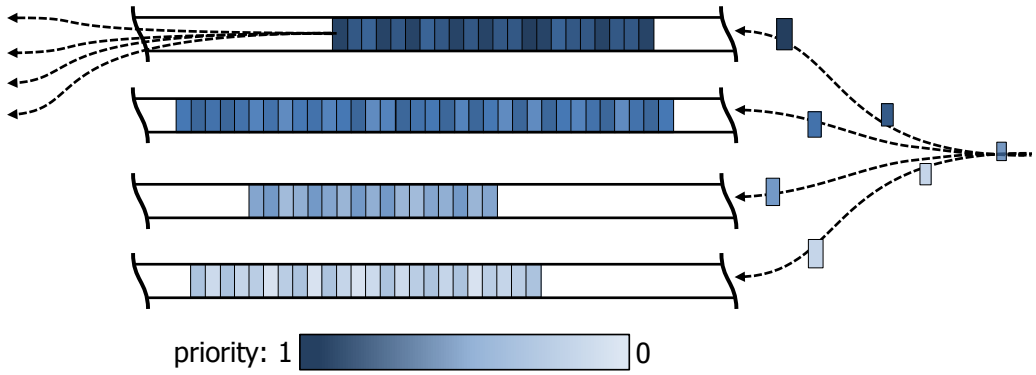


Figure 5.5.: Discretized priorities can be configured with very simple callback functions. During enqueue, the appropriate bucket is chosen, while dequeue takes tasks from the bucket with the highest priority that is not empty.

The corresponding callbacks are straightforward to set up:

```
template<int NumChildren>
struct Discrete {
  template<class Task>
  __device__ static
  int traverse(Task::Payload& payLoad){
    return payLoad.priority * NumChildren;
  }
  __device__ static
  int propagate(bool* infos) {
    for(int i = NumChildren-1; i >= 0; --i)
      if(infos[i])
        return i;
    return 0;
  }
}
```

```
struct LeafHasData {
  template<class Queue>
  __device__ static
  bool checkLeaf(const Queue& q){
    return q.count() > 0;
  }
};
```

`traverse` discretizes the priority and returns the id of the bucket with appropriate priority. `checkLeaf` determines if data is available in the queue, and `propagate` runs through the buckets in descending order of priority, choosing the first non-empty bucket.

5.3.2. Round-Robin

Another common strategy that is straightforward to implement is per-multiprocessor round-robin. Consider a setup with one root bucket and an arbitrary number of child buckets that should be accessed in a round-robin fashion. Every multiprocessor stores the id of the last child bucket chosen for dequeue in shared memory. During each invocation of the propagate method, the id is incremented to identify the next bucket for dequeuing.

```
1  template<int NumChildren, class Traverser>
2  struct RoundRobin : public Traverser {
3    __device__ static
4    int& getLast(){
5      __shared__ int last;
6      return last;
7    }
8    __device__ static
9    int propagate(bool* infos){
10     int &next = getLast();
11     next = (next + 1) % NumChildren;
12     for(int i = 0; i < NumChildren; ++i){
13       if(infos[next])
14         return next;
15       next = (next + 1) % NumChildren;
16     }
17     return 0;
18   }
19 };
```

Note that the Round-Robin class requires another class as template argument that is supposed to provide the `traverse` method. Deriving the scheduling policy from a custom template facilitates the reuse of existing Round-Robin mechanisms for dequeuing and mixing them with any kind of enqueue policy. The initial queue is chosen randomly through additional operations that have been omitted here for the sake of clarity. For detailed control, we support setting an optional initializer method that is called right after kernel launch.

5.3.3. Fair Scheduling

In a system that supports the execution of multiple processes, one common goal is to provide a fair load balancing setup to assign an equal amount of compute time to each process. In this context, a process can either be a single task that is respawned multiple times, or an entire group of different tasks that are capable of instancing each other. We consider two solutions for fair scheduling with our hierarchical bucket queue framework, by using either multiple, separate buckets or a single, sorted bucket.

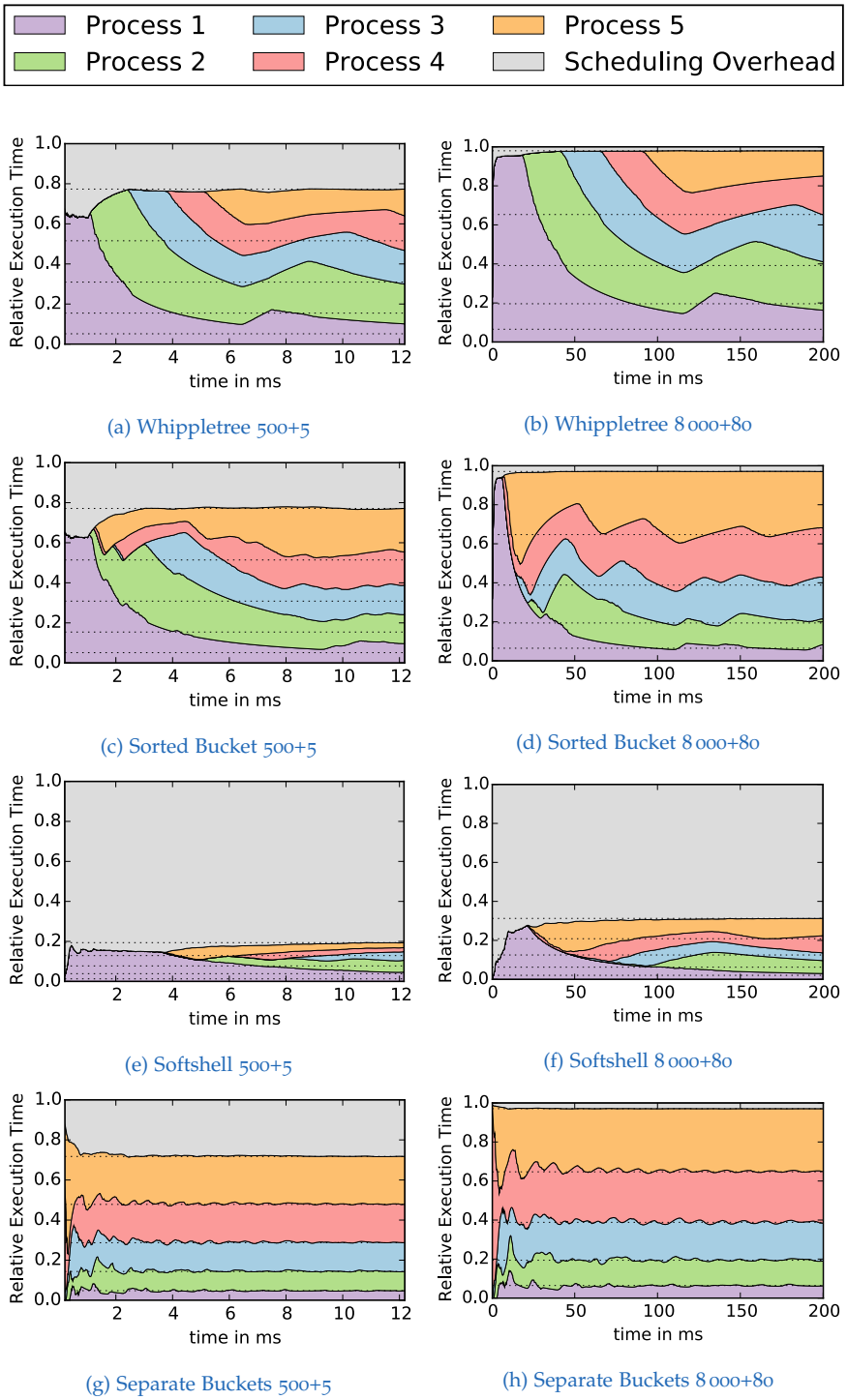
Separate Buckets The first way to implement fair scheduling in our system is by using separate child buckets that are pooled by a fair-scheduling root bucket. To implement this concept, we associate a counter with each child bucket and record the total time that processes from this bucket have consumed so far. During dequeue, the fair scheduling bucket selects the child whose counter is currently lowest. Keeping track of the total time consumed allows us to either assign equal compute times to all buckets or enforce predefined target quotas for individual processes.

Sorted Bucket Alternatively, quota-driven fair scheduling can also be set up by utilizing the queue maintainer. Mixing all tasks in the same queue, we can simply use the deviation in runtime from their desired target quota as sorting criterion. However, as sorting the queues takes time and the priorities are constantly changing, prioritization might significantly lag behind execution.

To measure the time spent on each process, we queried the built-in cycle counter present on each multiprocessor before and after executing a task. Note that this measure is not guaranteed to capture the exact time a task was actually executing instructions, as the hardware warp scheduler switches between all warps present on a multiprocessor based on their ready state. Thus, all tasks being executed on the same multiprocessor can influence the measured execution time for each other. If a more precise time measurement is needed, the megakernel can be configured to use a single, large thread block per multiprocessor. We can then make uniform scheduling decisions among the entire multiprocessor, and all tasks executed concurrently belong to the same bucket. In this case, the consumed clock cycles capture the time an entire multiprocessor was assigned to a certain process and the measurements can be considered fair.

Evaluation To evaluate both fair scheduling implementations and compare our approach with previous work, we set up five processes and launched 1 000 initial tasks for each. Each task was primed to execute a random number of fused-multiply-add (FMA) instructions and memory (MEM) operations to simulate diverse workload characteristics. In order to evaluate how behavior is influenced by the overall rate at which load balancing decisions need to be made, we set up two scenarios to generate different per-task loads: 500 FMA + 5 MEM and 8 000 FMA + 80 MEM on average. After executing, each task immediately enqueued a copy of itself to ensure that the system remained fully occupied. We recorded 10 000 scheduling decisions, capturing the time spent on each process. To measure the overhead of scheduling, we executed the same tasks with conventional CUDA kernel launches and recorded the difference to the average task throughput. We compared bucket queuing with Whippletree without scheduling and Softshell with priority sorting as references. The results are shown in Figure 5.6.

Results obtained from Whippletree indicate that handling light-weight tasks with 500 FMA and only a few memory transactions provide a challenging scenario for dynamic load balancing approaches. The overhead of storing and fetching task payloads in a queue led to a slowdown of about 20% in comparison with CUDA. However, in the 8 000+80 scenario, the overhead became negligible. Whippletree follows a simple FIFO approach and thus did not consider the desired quotas (dotted lines), assigning processing resources to one task after the other. Using the maintainer (Sorted Bucket) shows that progressive sorting caused hardly any overhead compared to Whippletree (less than 2%), which is not surprising, considering that only one out of 88 employed thread blocks was used for sorting on the Geforce GTX 980Ti. In the 500+5 scenario, sorting did not meet the quotas within 10 milliseconds, since tasks were consumed too quickly from the front of the queue to enable thorough sorting in the back. In the 8 000+80 scenario however, load balancing converged after approximately 100 milliseconds with noticeable oscillations around the target quotas. Softshell uses dynamic memory allocation for the payload and a hash map to combine payloads. Thus, its overhead is immense in comparison to a simple CUDA kernel (only 20% and 25% achieved throughput). However, as expected, its scheduling strategy (although slower) behaves similarly to the sorting implemented by the maintainer. Using separate buckets clearly achieved the best load balancing behavior in the examined scenarios. In comparison with Whippletree's FIFO execution, a slight increase in overhead occurs in both of our fair scheduling implementations, which we ascribe to the additional effort of checking multiple queues for dequeuing.



(a) Whippletree 500+5

(b) Whippletree 8 000+80

(c) Sorted Bucket 500+5

(d) Sorted Bucket 8 000+80

(e) Softshell 500+5

(f) Softshell 8 000+80

(g) Separate Buckets 500+5

(h) Separate Buckets 8 000+80

Figure 5.6.: Quota-driven scheduling with target time quotas (dashed lines) of 7%, 13%, 20%, 27% and 33%. Efficacy and overhead of our framework are compared against Softshell and Whippletree for reference. While separate buckets can quickly adjust the scheduling to match the desired quota, sorting takes significantly longer and oscillates around the target values.

5.3.4. Earliest-Deadline-First

Earliest-deadline-first is a common strategy in hard real-time scenarios, where all processing power is dedicated to the job with the closest deadline. Using our hierarchical bucket queuing framework, we find multiple ways to implement earliest-deadline-first scheduling. As our first approach, we can set up a maintainer that sorts tasks according to the deadline of their associated *job* (Sorted). Second, we simply use separate buckets for each job and always choose that with the earliest deadline (PerJob). Third, given an application that takes a known, finite amount of time to run, we can discretize the entire runtime into buckets (Discretized). Hence, each bucket corresponds to a specific time frame of the program's execution and tasks can be enqueued accordingly, while dequeue will always draw from the bucket with the closest upcoming deadlines. All tasks within a bucket must be executed before its associated time frame passes to ensure that no task deadline is missed.

In many applications, tasks that failed to meet their deadline can generally be skipped. Thus, as an optimization of our third approach, we can still work in discretized time but instead keep a ring buffer of buckets that only holds tasks up to a certain interval into the future (WrapAroundBuckets). As time progresses, the oldest buckets can then be reused for upcoming deadlines.

Evaluation To test earliest-deadline-first scheduling, we launched one controller block separate from the laboring megakernel to periodically create tasks for six task types at recurring intervals between $1ms$ and $8ms$. For each type, one to four tasks were created at every turn, each running between $0.1ms$ and $4ms$ to execute a mixture of FMA and MEM instructions. Each scheduling strategy was evaluated by gradually increasing the number of tasks by up to a factor of 128 of the initial load and measuring how many tasks still completed within their deadline. For WrapAroundBuckets, we used a future window of $10ms$ and 256 buckets. Again, we compare overall behavior and performance for each of our hierarchical bucket queuing solutions to Whippletree and Softshell. The results for these tests are shown in Figure 5.7.

Except for Softshell, all approaches managed to respect deadlines for low workloads. Unfortunately, Softshell's overhead was simply too high to cope with a load multiplier above 1. Tasks were issued periodically one interval's time before the deadline. Thus, their occurrence, to a certain extent, followed the deadline and simple FIFO scheduling (employed in Whippletree) with its lower overhead could keep up with the other approaches.

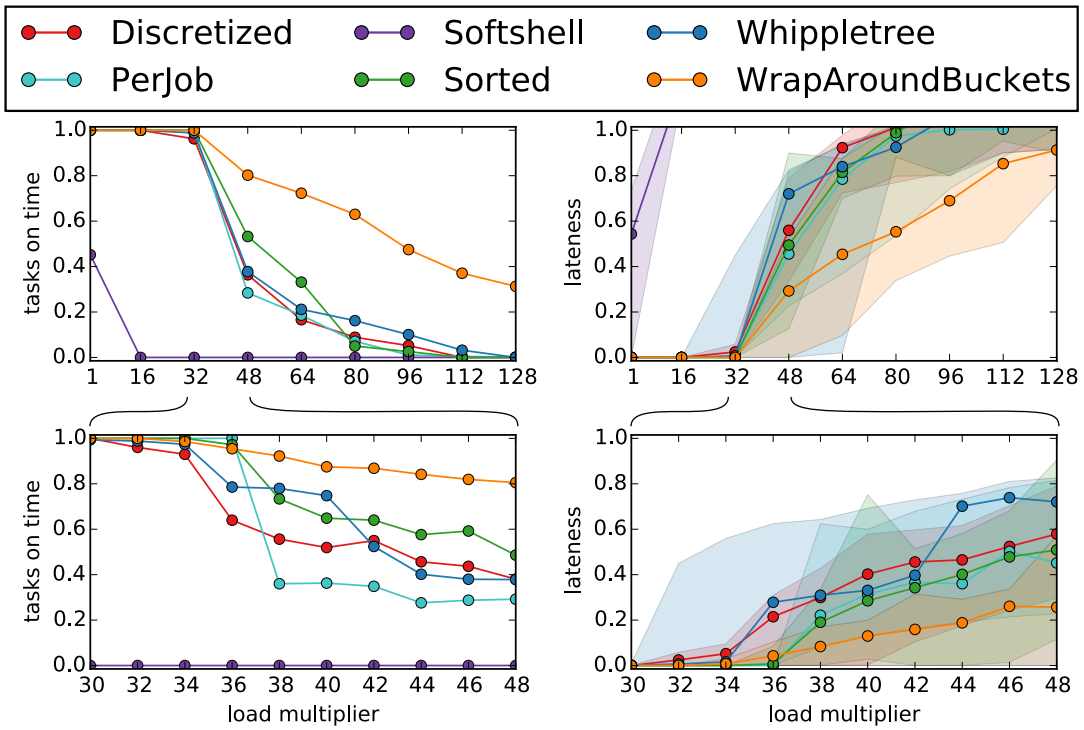


Figure 5.7.: Earliest-deadline-first results for different scheduling implementations with detailed closeups on the bottom. (left) The fraction of tasks executed on time by each method as the workload increases. (right) The average lateness shows by how much time on average tasks miss their deadline. The minimum and maximum lateness for each method are overlaid as well.

However, at higher loads (starting at a load multiplier of about 30), differences emerged. Whippletree and Discretized did not (or could not) identify the job with the closest deadline and were the first approaches to miss deadlines. Soon after, PerJob and Sorted also dropped in performance. PerJob's drop is comparatively steep, which can be explained by the fact that PerJob still tries to schedule tasks that already missed their deadline. Also, PerJob showed a relatively high overhead, which we ascribe to the necessity of finding the queue with the earliest deadline to dequeue items. However, the average lateness increased similarly for Discretized, Whippletree, PerJob and Sorted. WrapAroundBuckets performed better than all other approaches, with comparatively high ratio of tasks on time and exhibiting a smaller increase in average lateness with increasing load. It is the only approach that clearly outperformed the simple FIFO scheduling implemented by Whippletree.

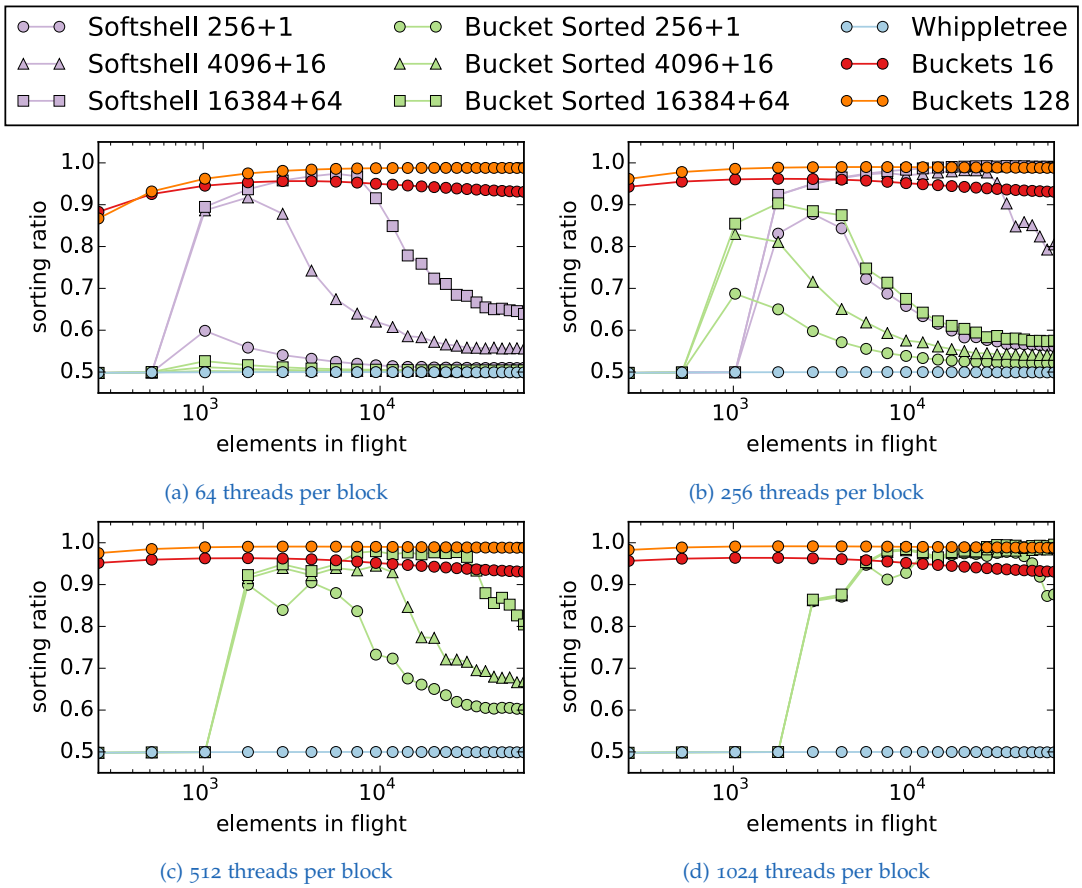
5.3.5. Application Defined Priorities

Finally, we evaluated the capabilities of bucket queues in applications that involve tasks with arbitrary priorities. A possible example for such a scenario can be given by any adaptive algorithm where the importance of a particular operation is difficult or impossible to predict in advance, *e.g.*, REYES-style subdivision or adaptive image sampling. We compare the performance of queue sorting to discretized-priority buckets. In our test setup, we launched W initial tasks and assigned uniformly distributed, random priorities to them. Every task executed a variable workload of FMA and MEM operations and spawned another task with a random priority. This way, an average of W tasks were contained within the queuing structure at all times during the evaluation. We recorded the order in which tasks were executed, as well as their associated priority. The test was stopped as soon as N tasks had finished. We computed the achieved scheduling accuracy as

$$S = \frac{1}{N(W-1)} \cdot \sum_{i=0}^{N} \sum_{j=0}^{W} s_{i,j},$$

where i and j may represent any two tasks that were concurrently queued for execution. $s_{i,j} = 1$ for tasks i and j, iff the task with the higher priority was chosen first; otherwise, $s_{i,j} = 0$. Note that the restriction to simultaneously enqueued tasks is necessary for a meaningful assessment, since a high-priority task n that was only generated after a low-priority task m finished could not possibly be processed before m. For random priorities, no scheduling at all leads to an expected accuracy of approximately 50%. If all elements are executed in the correct order, scheduling accuracy equals 100%. As multiprocessors execute in parallel, 100% accuracy may never be reached in practice.

Evaluation The measured scheduling accuracy is shown in Figure 5.8. As reference for comparison, we again include Softshell and Whippletree. Softshell and Bucket Sorted exhibited similar behavior. Queue sorting turned out to be ineffective under low and high loads. With only a few elements in the queue, sorting was not able to start as there was no sufficient safety margin and thus the achieved accuracy was about 50%. On the other hand, with a high number of elements in the queue, progressive sorting was not fast enough to move high priority elements to the front. Consequently, sorting accuracy quickly deteriorated with rising number of elements in flight W.



(a) 64 threads per block

(b) 256 threads per block

(c) 512 threads per block

(d) 1024 threads per block

Figure 5.8.: Scheduling accuracy with varying threads per block and task complexity: (Softshell|Bucket Sorted) [FMA]+[MEM]. Bucket queues used 16 or 128 buckets.

The comparably high scores recorded for Softshell are misleading; due to its immense scheduling overhead, a lot more time could be spent on sorting relative to time spent on task execution and thus higher accuracies were achieved for the same W. The performance of both sorting techniques is also dependent on the task duration, since long-running tasks (*e.g.*, 4096+16 or 16394+64) are dequeued less frequently, and, thus, remain longer inside the queue and undergo additional sorting passes. In contrast, the accuracy of Buckets was not affected significantly by task runtime, nor by W. We observed a slight drop in performance for very large W. Since discretized prioritization leads to fast consumption of high-priority tasks, a larger number of tasks accumulated in low-priority queues and could not be accurately distinguished. Overall, the error of bucket queues approximately equaled the expected discretization error. With 16 buckets, we achieved an accuracy

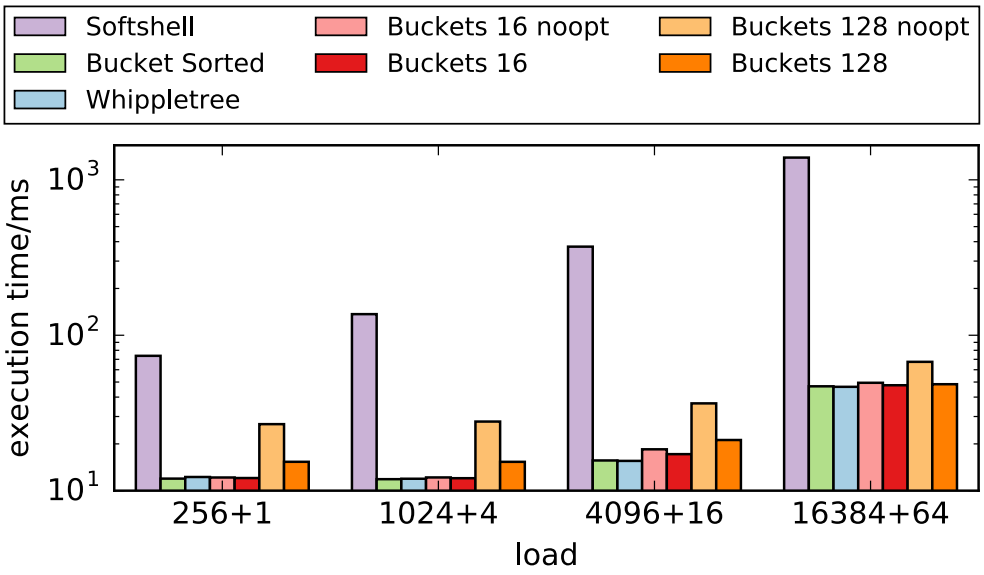


Figure 5.9.: Softshell shows up to 10× the execution time of the other approaches. Bucket Sorted and Buckets 16 show only a small execution time overhead. Using a high number of buckets without our upward propagation optimization significantly increases execution time (up to 2×).

of up to 94%; with 128 buckets, this figure rose to 99%. Bucket Sorted only achieved such high accuracies with more than 512 threads being used for sorting, and about 10 000 elements in the queue. Since Whippletree does not consider per-item priorities in its scheduling at all, it yielded an expected default accuracy of 50%. Softshell failed to run to completion for 512 and 1024 threads per block.

The impact on execution time is shown in Figure 5.9. The log-scale plot demonstrates that Softshell took roughly ten times longer than all other techniques to finish any of the test cases. Bucket Sorted again only added a small overhead to the execution time compared to Whippletree. As Buckets did not run a sorting algorithm, it conserved more bandwidth and processing power for task execution. However, with an increasing number of buckets, more time was spent on traversing the bucket hierarchy. Processing the hierarchy top-down with only a single thread (Buckets noopt) instead of bottom-up (see Section 5.2.4) more than doubled the execution time (256+1 and 4096+16). Using our bottom-up approach (Buckets), the overhead was significantly reduced, underlining the usefulness of this design choice in the first place. The 16 bucket version managed execution times on par with Whippletree while achieving a scheduling accuracy of up to 94%.

5.4. Implementing Adaptive Rendering

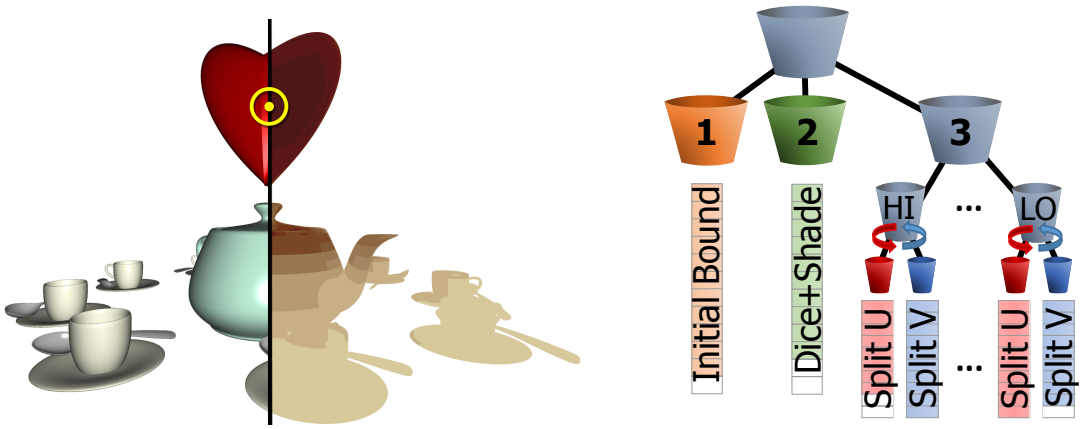
In order to demonstrate how hierarchical bucket queuing is applicable to rendering, we present two use case scenarios: micropolygon rendering and path tracing. For both applications, we assume soft real-time constraints.

5.4.1. Foveated Micropolygon Rendering

Virtual reality applications demand low-latency, high-resolution image synthesis, especially when using an HMD output device. Frame latency is a particularly sensitive issue in such applications and must be kept below a certain threshold. With fast eye movements, it is usually difficult to predict rendering times, as the focus might quickly change from object to object, leading to the violation of latency constraints. One potential way to address this problem would be to use a rendering method that allows gradual refinement of the image until the deadline for maximum tolerable latency expires, thus maximizing the image quality within a given timeframe.

One rendering technique that produces such gradual refinement is micropolygon rendering, as used in the REYES pipeline (Cook, Carpenter and Edwin Catmull, 1987). While REYES-style micropolygon rendering has been implemented on the GPU before (Tzeng, Patney and Owens, 2010; Steinberger, Kenzel, Boechat et al., 2014), using our prioritized load balancing, we can now take advantage of foveated rendering (Cater, Chalmers and Ledda, 2002) to significantly improve image quality by prioritizing refinement around the user's fixation point. Using dynamic priorities, we are able to set up foveated rendering in a way that both satisfies latency deadlines and generates high quality in focus areas.

The central task of micropolygon rendering is posed by the recursive bound-and-split loop to subdivide patches until, eventually, they are small enough to be shaded as a grid of micropolygons. Ideally, we want a smooth transition where fine and coarse patches meet and reasonable visual quality throughout the entire image to avoid popping artifacts during motion. For each patch, we evaluate its split priority based on its projected patch size and distance to the current fixation point, thus prioritizing large patches that are close to the fixation point. Dynamic load balancing is essential in this example, since we cannot predict the projected extents of all patches resulting from recursive



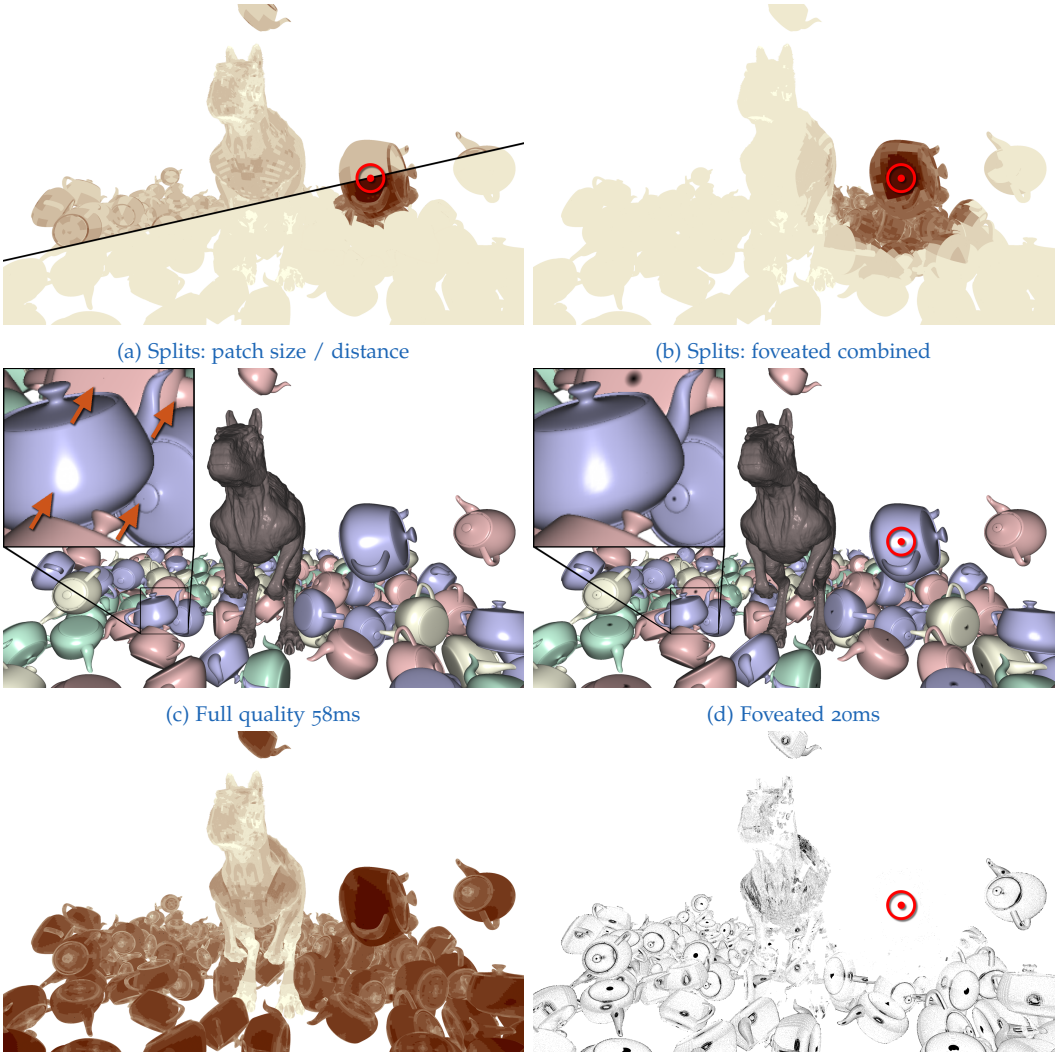
(a) Foveated rendering of Tea scene

(b) Micropolygon render setup

Figure 5.10.: Foveated micropolygon rendering. (a) The color intensity on the right side indicates the degree of subdivision applied, which is highest at the fixation point (yellow ring mark) and gradually falls off with increasing distance. (b) Schematic visualization of the bucket queue setup to achieve the desired prioritization.

splits in advance. Building on the Whippletree REYES implementation (Steinberger, Kenzel, Boechat et al., 2014), we set up a three-level bucket hierarchy, as shown in Figure 5.10b. The first bucket distinguishes between split tasks and all other task types. It prioritizes the execution of the initial bound tasks, shading and dicing over patch splits. This leads to an early creation of all initial split tasks and drains the pipeline of all intermediate data that is ready to be rendered. On the second level, split tasks are inserted into discretized priority buckets. Below each priority bucket, a round-robin scheduler switches between splits along the u and v direction, which are implemented as separate tasks. Before we enqueue patches to be split, we check if the procedure is likely to execute before the target latency is reached. If this is not the case, we stop the recursion and directly forward the patch to the dicing and rendering stages. For time measurement we again rely on the per-multiprocessor cycle counts, which we synchronize in each frame.

We test our approach with two animated scenes: Killeroo and Tea, shown in Figures 5.10a and 5.11. For rendering, we chose a viewport with a 4K resolution. Conventional REYES rendering of the Killeroo scene at full image quality takes between 30ms and 60ms, the Tea scene takes between 15ms and 22ms. With foveated rendering, we can limit the Killeroo scene to a guaranteed 20ms and the Tea scene to 10ms by adaptively generating full image quality only around the fixation point and lower quality in the remaining image. Figure 5.11 demonstrates the details of our approach for the Killeroo scene.



(a) Splits: patch size / distance

(b) Splits: foveated combined

(c) Full quality 58ms

(d) Foveated 20ms

(e) Splits: full quality

(f) difference enhanced 200×

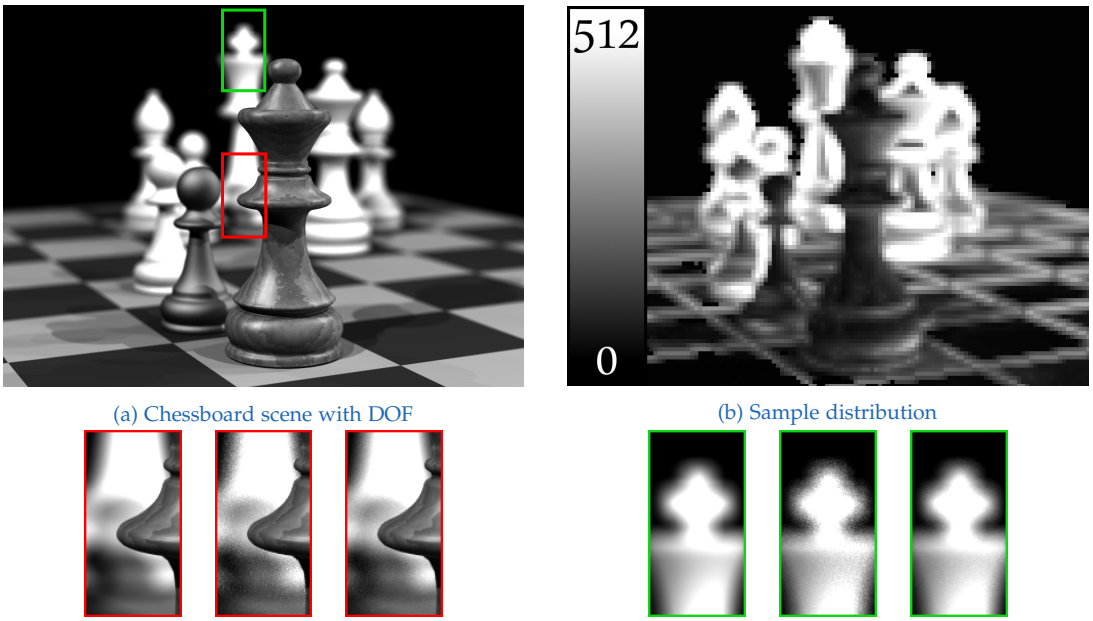
Figure 5.11.: (a, top) Using the projected patch size as priority, the processing power is distributed evenly; (a, bottom) using the distance to the focus point as priority instead leads to very localized refinement. (b) A combination of both creates a reasonable falloff. (c and e) A full-quality render pass produces high geometric detail, but requires a considerable amount of time. Using foveated rendering, only the focus area (red ring mark) is rendered at full quality and image synthesis is sped up significantly. (f) Visualization of the difference image between full quality and foveated rendering.

5.4.2. Adaptive Sampling for Path Tracing

While Monte Carlo path tracing on the GPU is becoming increasingly popular, generating high-quality images quickly remains a challenging task due to the notorious noise, which is most prominent in the early stages of image synthesis. For implementations on the CPU, adaptive, priority-based solutions have been proposed to reduce noise and enhance image quality with a low sample budget, *e.g.*, by distributing more samples to image regions with a high estimated local error (Hachisuka et al., 2008; Overbeck, Donner and Ramamoorthi, 2009). Since the error estimate and the rate of convergence in an image region may change with each new sample, adaptive sampling usually relies heavily on a continually maintained priority queue to always identify the region with the highest error estimate before casting new samples.

Porting the adaptive sampling approach to the GPU is non-trivial, since massively parallel execution generally provides little support for efficient and adequately sorted queues. However, with bucket queue hierarchies, different prioritization schemes can be incorporated efficiently. As a demonstration, we implemented a path tracer for static scenes, with support for low-discrepancy sequence sampling, depth-of-field (DOF) and an arbitrary number of light sources. Our implementation uses a single task type, which casts four rays with a random number of maximum bounces for each pixel upon execution. To avoid write conflicts, each task is assigned to a dedicated 8×8 pixel region of the output image. Upon completion, each task enqueues a copy of itself, thus allowing the persistent thread blocks to constantly fetch new work for execution. This approach has been shown to achieve high occupancy on GPUs (Aila and Laine, 2009). We force the persistent threads execution to pause every $50ms$ to display the current image via OpenGL. Using this one task in Whippletree establishes our base-line implementation, which, in practice, performs uniform sampling of the image domain.

In order to enable adaptive sampling through prioritization, we set up a single-level hierarchy with 128 individual buckets. Furthermore, a current local error needs to be computed before enqueuing. Given an approximate, predefined lower and upper bound on the error for image synthesis, task priority is obtained by interpolating the current local estimate between bounds: high error equals high priority. Naturally, a rendering algorithm cannot depend on high-level knowledge of the final image in order to compute the difference to the ground truth. Instead, we can collect information from cast samples to compute an error estimate based on per-pixel variance.



(a) Chessboard scene with DOF

(b) Sample distribution

(c) Mixed-focus region

(d) Out-of-focus region

Figure 5.12.: Comparison of uniform and adaptive sampling using the expected gain error estimate as priority. (a) Path-traced chessboard scene with DOF and 9 light sources rendered at resolution 1024×750. (b) Sample distribution with prioritization after 4*s*, brighter means more. (c, d) Comparison of the result of uniform and adaptive sampling after 4*s* to ground truth (2048 samples/pixel) in marked regions. Left: ground truth; Center: uniform; Right: adaptive sampling.

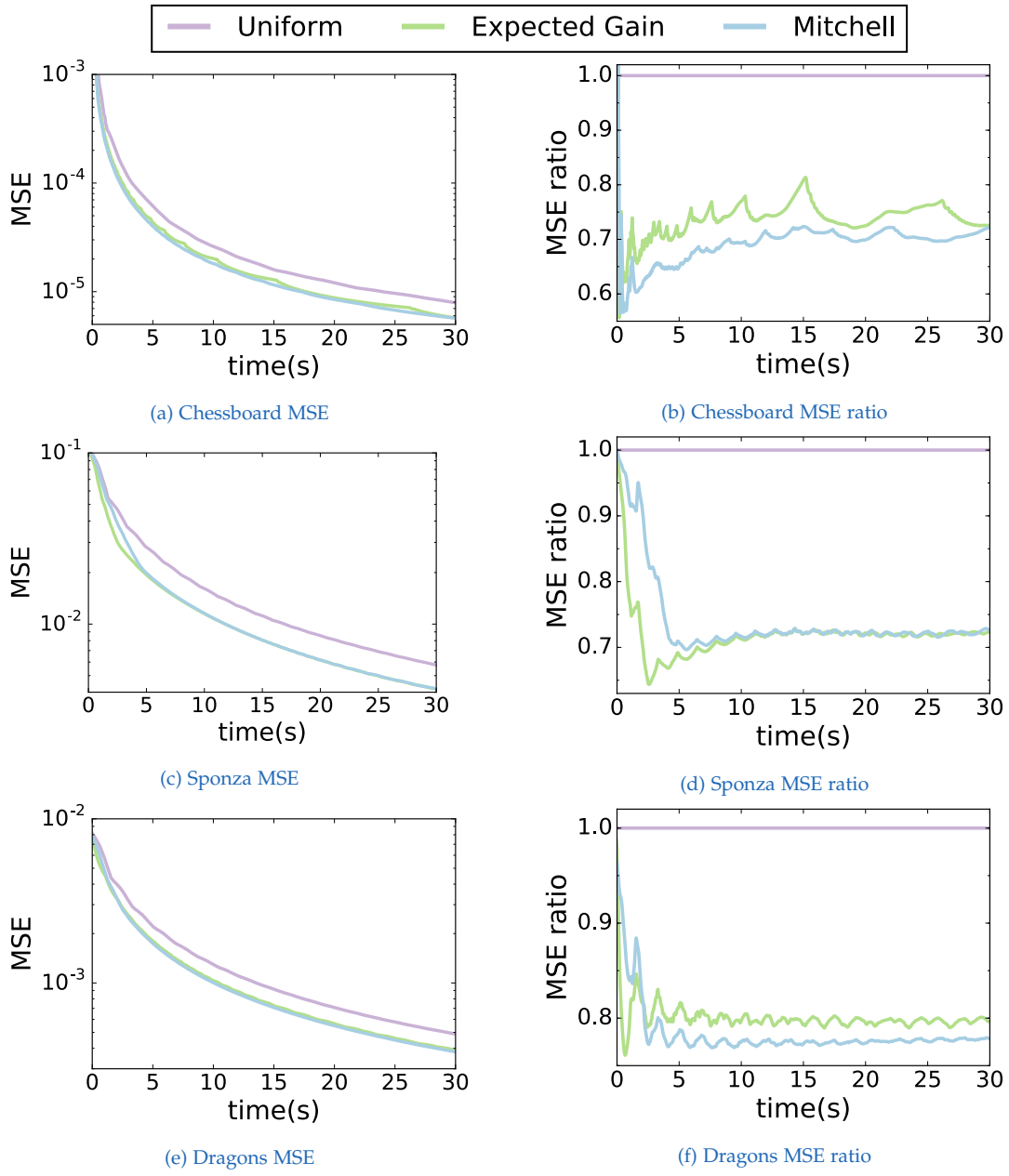
To evaluate our approach, we test two different per-pixel error metrics. First, we use the expected error reduction constructed around the Monte-Carlo error estimate $E \propto \sigma/\sqrt{N}$:

$$\Delta E \approx \frac{\sigma}{\sqrt{N}} - \frac{\sigma}{\sqrt{N+4}},$$

where σ is the current variance estimate over all N samples for a pixel so far. We call this error metric 'expected gain'. Second, we use the error metric devised by Mitchell (1987) for obtaining anti-aliased images. Per-pixel estimates are added up to obtain the local error estimate for an 8×8 region. Both metrics require at least two samples to compute an initial variance estimate. Thus, we set the priority of the initial tasks to the maximum, forcing four samples to be computed for each pixel before relying on prioritization for further sampling. Figure 5.12 outlines the adaptive behavior when using the expected gain metric for sampling the scenes. At equal run times, our prioritized rendering reduces noise in comparison with uniform sampling.

In Figure 5.13, we show the development of the mean squared error (MSE) over time for the two prioritization schemes after the initial four samples have been computed. Two additional scenes (Sponza and Dragons), along with the sampling distribution generated by our algorithm, are shown in Figure 5.14. Adaptive sampling quickly reduces the MSE by up to 45% for the Chessboard scene, 35% for the Sponza scene and 22% for the Dragons scene when compared with uniform sampling. Note that the MSE is only recorded once all image regions have been sampled at least once. Any deviations from uniform sampling before that cannot be attributed to prioritization, since error estimates cannot be computed for unsampled regions. Expected gain metric seems to perform slightly better at early frames, but loses its advantage after the initial noise has been cleared up.

Decreasing Priorities It should be noted for both applications that, as the rendering procedure progresses, the overall geometry coarseness or the estimated error for image regions is bound to continuously decrease. Consequently, since the placement in a priority bucket is computed via linear interpolation between defined minimum and maximum bounds, eventually all tasks will be repeatedly placed in the lowest priority bucket queue, thereby prohibiting prioritization. While this did not pose an issue in our evaluation due to its short duration dictated by the soft real-time constraint, it should be considered for general adaptive systems. One solution would be to use a non-linear function for computing the queue index to emplace patches or image regions, but this may not suffice. A more general approach would be to adapt the upper bound for interpolation, depending on the fill levels of the queues. Since our implementations always dequeue from the first non-empty bucket with top priority, we can track how many higher-priority queues were skipped due to them being empty. Hence, if a given number of buckets at the top of the priority range is continuously skipped for a substantial amount of time, it means that they are going unused. In that case, the upper interpolation bound used for computing the appropriate bucket index could simply be reduced by a scaling factor, in order to reactivate currently unused priority queues. Naturally, doing so would require that all tasks in this hierarchy need to be reclassified once with the new priority function before priority queuing as usual can resume. For micropolygon rendering, all threads would have to take patches from the hierarchy until they agree that it is empty, then process them from a transition queue until they agree that it has been emptied as well. For adaptive sampling, we can simply use atomic counters, since the number of queued image regions is determined by the resolution and therefore fixed.



(a) Chessboard MSE

(b) Chessboard MSE ratio

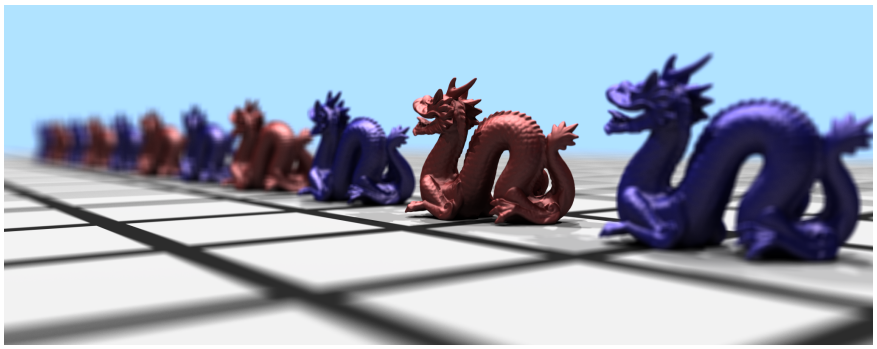
(c) Sponza MSE

(d) Sponza MSE ratio

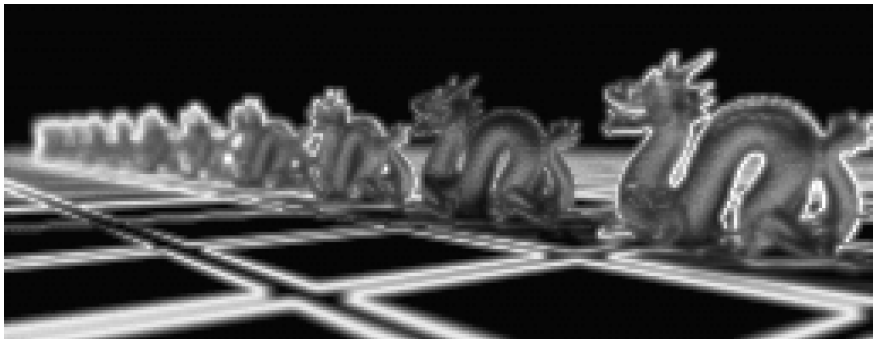
(e) Dragons MSE

(f) Dragons MSE ratio

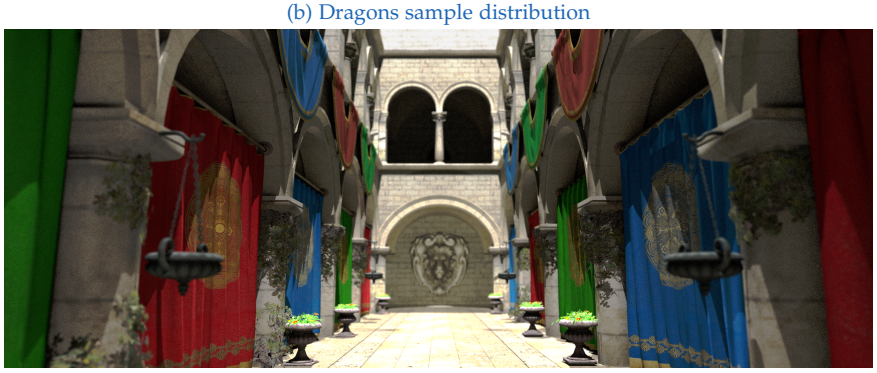
Figure 5.13.: Progression of the MSE during path tracing. (a, c, e) With priority scheduling, the MSE reduces more rapidly. (b, d, f) The ratio of the MSE relative to baseline uniform sampling shows that the MSE is reduced by up to 45% in early frames.



(a) Dragons scene



(b) Dragons sample distribution



(c) Sponza scene

(d) Sponza sample distribution

Figure 5.14.: Adaptively sampling two scenes with hierarchical bucket queuing

5.5. Remarks on Load Balancing and Rendering

Using hierarchical bucket queues, it is, for the first time, possible to set up a variety of priority-based load balancing policies directly on the GPU. Bucket queues can be tailored to the needs of the application, not only for priority scheduling, but also to combine tasks or integrate multiple processes into a single scheduler. Bucket queues are easily configured by defining a small number of callback functions. They enable simple load balancing schemes, like FIFO or round-robin scheduling, as well as more complex strategies, like earliest-deadline-first or fair scheduling. According to our tests, prioritized load balancing using hierarchical bucket queues clearly outperformed previous sorting-based or coarse-grained approaches, making it the preferred choice for scheduling on the GPU. We integrated our bucket queues into the Whippletree framework and were able to show that even in challenging situations, our approach only adds a small overhead (between 1% to 5%) compared to the original framework's baseline. While being directly applicable to any persistent threads approach, our design also lends itself well to a potential implementation in hardware.

As a classic example of GPU workload, rendering procedures can also benefit from priority scheduling, as we have demonstrated for latency-controlled, foveated micropolygon rendering and priority-driven path tracing. Foveated rendering is enabled by the ability to direct the available processing power to a particular focus region. With path tracing, prioritizing the image areas that are expected to reduce the image error the most can quickly decrease the difference to the ground truth by up to 45% in comparison to simple uniform sampling. While priority-based rendering is certainly of interest for future virtual reality systems, it will require additional work to integrate priorities deeper into the rendering pipeline.

Although hierarchical bucket queuing provides a general solution to dynamic load balancing for software rendering, as of now, it can only be applied to the GPU when seen as a fully programmable, general-purpose co-processor. However, load balancing is also of great importance when we consider other aspects of the GPU, specifically its role as a high-performance sort-middle rasterization pipeline. Doing so naturally restricts the degree to which we can manipulate GPU behavior, since multiple integral features—such as built-in load balancing mechanisms—are intentionally concealed out of necessity or to satisfy confidentiality.

Nevertheless, since we know the fundamental principles of its architecture, we can speculate that built-in load balancing mechanisms must exist in order to achieve high efficiency. Revealing those mechanisms could help us optimize our input to the pipeline, provide us with insights into guiding hardware design directives and furthermore inspire concepts for future, elaborate GPU software rendering solutions.

6. Batch-based Load Balancing: Vertex Reuse and Optimization

Contents

The same transformed vertex is on average required six times during rendering of a typical triangle mesh. To avoid redundant vertex transformation, graphics processors have traditionally employed a *post-transform cache*, often simply referred to as the vertex cache. The assumption of this module in hardware sparked research into the topic of how to optimize geometry to maximize locality of vertex references for caching. With the increasingly massive parallelism found in modern GPUs, the idea of the geometry stage relying on a centralized cache structure is now at odds with the tenet of load balancing and has recently been challenged (Barczak, 2016; Giesen, 2011). However, no detailed analysis beyond faint suspicions has been published on this matter. In this chapter, we put the hypothesis of graphics hardware using a post-transform cache to the test. We design and conduct experiments that clearly show that the behavior of vertex processing on a modern GPU is not consistent with the assumed presence of a traditional post-transform cache.

6.1. Optimizing for the Post-Transform Cache

Hoppe (1999) notably suggested the use of a post-transform cache for graphics hardware and demonstrated their now widely-known mesh processing algorithm to generate cache-optimized triangle strips from input geometry data. In order to maximize the cache hit rate, triangles are reordered by greedily growing strips and introducing cuts whenever the strips become too long to still effectively exploit the cache for vertex transformation. Hoppe's algorithm was included in the D3DX library from DirectX 9 onward and has since been migrated to the *DirectXMesh* project, which remains a popular tool for developing graphics applications. Lin and Yu (2006) later developed another algorithm, *K-Cache*, which iteratively selects focus vertices and outputs all connected faces, before marking the focus vertex as visited. Forsyth (2006) also presented their own mesh optimization algorithm, which assumes an exponential falloff for the probability of a cache hit. Another fast, cache-based mesh optimization algorithm is *Tipsify* (Sander, Nehab and Barczak, 2007), which is part of AMD's Triangle Order Optimization Tool (*Tootle*).

Despite these efforts, detailed evidence for the actual implementation of a post-transform cache and the associated benefit from cache-oriented mesh optimization algorithms on modern GPUs is scarce, driving researchers to adopt microbenchmarks for exposing actual hardware behavior (Barczak, 2016; Jia et al., 2018). The lack of mention of a post-transform cache in recent articles (Purcell, 2010; Kubisch, 2015) raises the question to which degree this widely-accepted concept still aligns with the reality of these devices. As demonstrated by Kenzel, Kerbl, Tatzgern et al. (2018), a batch-based approach seems to match the behavior of modern GPUs much more closely, confirming our intuition that the primary objective on massively parallel architectures should be efficiently dividing and distributing incoming workload, rather than minimizing the number of redundant vertex computations. Through thorough testing and analysis, we are able to reveal in this chapter some of the details for actual reuse strategies that current GPUs seem to follow. Armed with the knowledge gathered in these reverse-engineering attempts, we are able to predict how GPUs will subdivide an input stream into independently processable batches with great accuracy. We present novel insights on how non-trivial load balancing in the hardware pipeline is achieved under the constraint of adequate vertex reuse. We then turn our attention towards the topic of mesh optimization and outline a general algorithmic framework for reordering indexed triangle sets for batch-based vertex processing, given that the concrete batching routine used by the target hardware is fully understood.

6.2. GPU Vertex Reuse Strategies

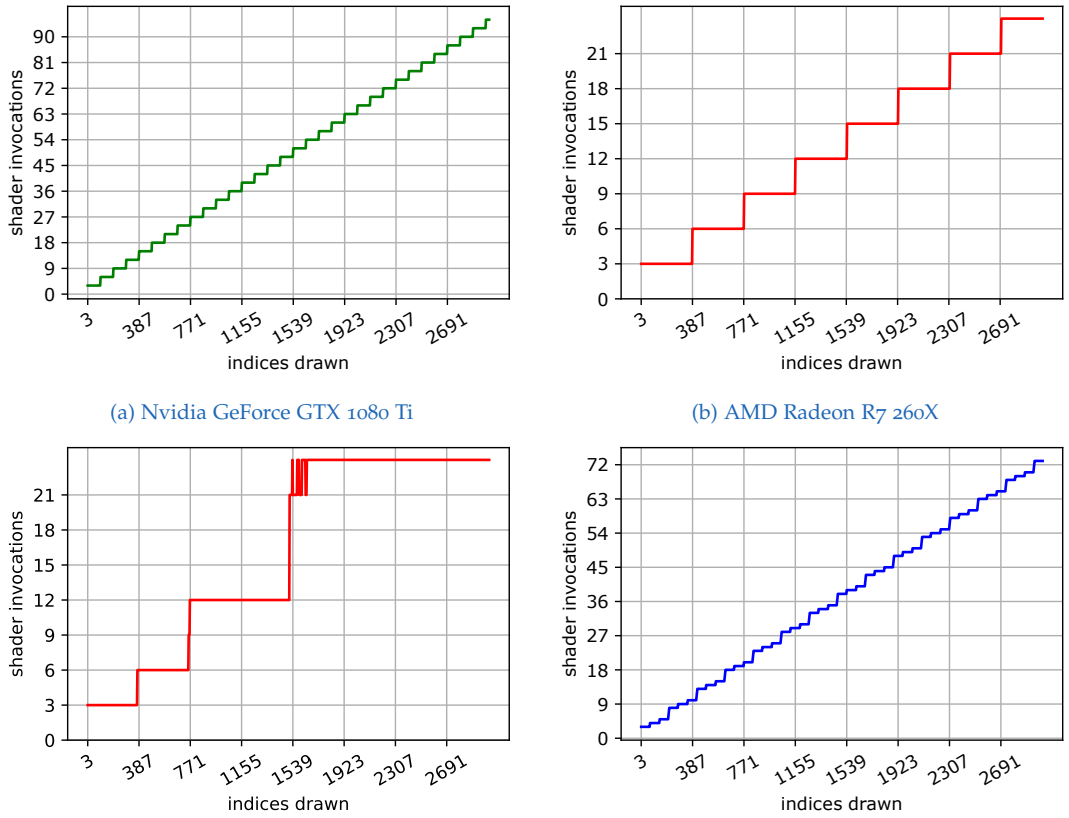
To go about answering the question whether a modern GPU uses a traditional vertex cache or not, we conduct the following experiment: We set up a vertex buffer holding three vertices and an index buffer containing the sequence $\{0, 1, 2, 0, 1, 2, 0, \ldots\}$ up to a given maximum length. We then draw increasingly larger portions of this sequence. Using a Direct3D 11 pipeline statistics query, we record the number of times the vertex shader is invoked when drawing each portion. To account for effects related to the size of transformed vertex data, we repeat the experiment for different numbers of output vertex attributes, ranging from 0 to the maximum 124 supported by Direct3D.

Figure 6.1 shows the behavior of the vertex shader invocation count as the number of indices drawn increases for a selection of GPUs, which are representative of the behavior we have observed for different vendors. Assuming the presence of a common post-transform cache of reasonable size, for this input stream, we would expect each vertex to be transformed exactly once. However, on almost all GPUs that we tested, we observe the vertex invocation count rising continuously as the stream length surpasses certain thresholds. The only exception we have seen stem from older Intel GPUs, which did, in fact, exhibit the behavior expected of a post-transform cache. This is consistent with publicly available documentation for these models (*Developer's Guide for Intel® Processor Graphics For 4th Generation Intel® Core™ Processors* 2013).

To summarize our findings so far: None of the assessed modern GPU models seem to employ a singular post-transform cache. The number of output vertex attributes does not seem to affect any GPU's ability to take advantage of vertex reuse. Furthermore, we can also rule out the presence of multiple independent post-transform caches instead of a single shared one, as such an architecture would lead the vertex invocation count to eventually plateau. However, even when drawing up to 30 million triangles, the vertex invocation count kept consistently rising on all tested, modern GPU models. Hence, in order to identify actual GPU vertex reuse strategies, further assessment is required.

6.2.1. Measuring Vertex Reuse

To quantify vertex processing efficiency, we introduce the *average shading rate* (ASR) measure, *i.e.*, the average number of vertex shader invocations per triangle drawn. An analogous measure that has been commonly used in the



(a) Nvidia GeForce GTX 1080 Ti

(b) AMD Radeon R7 260X

(c) AMD Radeon RX Vega 56

(d) Intel HD Graphics 520

Figure 6.1.: The number of times the vertex shader is invoked on a selection of GPUs from different vendors when drawing a recurring $\{0, 1, 2\}$ index sequence of increasing length. A post-transform cache of reasonable size would achieve perfect reuse in this scenario, processing every vertex exactly once and leading to a total of three shader invocations, independent of the input sequence length. The observed 'staircasing' is indicative of a batch-based approach for processing of the input primitive stream, where vertex reuse is only possible locally within each batch.

past is the *average cache miss rate* (ACMR). However, in light of our findings, we would argue that thinking about vertex reuse in terms of caching is not necessarily beneficial. In order to measure the ASR achieved on a particular GPU for a given triangle mesh, we can use the same approach as before where we relied on a Direct3D 11 pipeline statistics query that allows us to measure the total number of vertex shader invocations. On more recent hardware, a more fine-granular result can be obtained by using atomic operations to increment a per-vertex counter from within the vertex shader.

6.2.2. Collecting Detailed Batching Information

Given the massively parallel nature of a modern GPU, it can be assumed that the apparent failure to reuse indefinitely repeating vertex data is caused by subdivision of input into smaller, independent batches to accommodate its large number of concurrent shading units. To understand the formation process of batches by GPUs to consolidate the necessity for load balancing and vertex information reuse, we aim to examine which vertices are processed by individual thread groups. Since we consider multiple vendors in this chapter, we use the general term *wavefronts*, or simply *waves* instead of *warp*.

Luckily, experimental support for Shader Model 6.0 (recently added to the Direct3D 12 API) exposes wave-level operations that allow cross-communication between threads that share a wavefront. Although not officially supported for vertex shading, we found that they can, in fact, be used even then, at least on all GPUs with Direct3D 12 that we tested. With the help of these cross-communication commands, we are able to write a vertex shader that outputs the information of exactly which vertex indices are processed in parallel within the same wave. We start by determining how many threads are currently running the vertex shader in a given wave. The thread with the lowest lane id then atomically increments a counter to allocate space in an output buffer and communicates the start index to all other threads in the wave. Using a parallel prefix sum, we assign a unique index to each active thread. Finally, each thread writes the vertex index they are currently assigned to process into the output buffer at the given offset. The first thread additionally outputs the total number of vertex indices processed by the wave.

6.2.3. Identifying Batch Patterns and Boundaries

We describe here a set of reproducible experiments that we used to confirm the trends and patterns we observed from the collected batch information. Their description also serves to outline the logic and constraints for forming batches on Nvidia and AMD GPUs from recent generations with practical examples. The results from our initial assessment give us a starting point to further investigate batching behavior: The locations at which steps for the vertex invocation count occur actually give us the first indicator, namely the maximum size of batches in an ideal case (*i.e.*, perfect reuse throughout the mesh). For the sake of brevity, we will refer to this parameter henceforth as MAX_SIZE. For Nvidia, we can consistently observe steps every 96 indices

(see Figure 6.1a). This makes sense, in so much as we know warps on Nvidia to run 32 threads in lockstep. Thus, in an ideal scenario, a batch size of 96 indices allows every thread to process a separate triangle, defined by 3 different indices. On the AMD R7 200, we find the MAX_SIZE to be 384 (see Figure 6.1b). It should be mentioned that, on the RX Vega 56 (Figure 6.1c), this figure seems to vary, relative to the number of indices in the index buffer, peaking at 384 as well. We have not investigated this behavior further, as even the smallest of our test cases triggers the maximum batch size of 384 to take effect. We confirm these numbers by drawing from an index buffer where each assumed batch is filled with a different, repeating triplet of unique indices. For a MAX_SIZE of 6, *e.g.*, the index buffer would be constructed as $\{0,1,2,0,1,2 \mid 3,4,5,3,4,5 \mid 6,7,\ldots\}$. On Intel models HD 520 and HD 630, we found that vertex reuse seems to be more dynamic, largely depending on the number of different indices being rendered. For the repeating test sequence of 3 unique indices, full reuse only happens within 66 indices and partial reuse up to 208 (see Figure 6.1d); however, for other data which we will describe shortly, we have found a maximum reuse window of up to 1791 indices.

Next, we vary the data within each assumed batch of length MAX_SIZE, to see if other limitations apply. For this, we can fill the index buffer with a consistently increasing sequence of N indices, where N is divisible by 3 and $N < MAX\_SIZE$, followed by $(MAX\_SIZE - N)$ indices in triplets $\{N-4, N-3, N-2\}$. Simply put, the index buffer contains $\frac{N}{3}$ unique triangles and then repeats the second-but-last triangle until we hit MAX_SIZE. Assuming that vertex reuse in a batch works at least over two consecutive triangles, the average shading rate must, therefore, remain at $\frac{N \cdot 3}{MAX\_SIZE}$ as long as the batch goes on. On Nvidia, we found that this condition failed at $N = 33$ and the ASR jumped to $\frac{(N+1) \cdot 3}{MAX\_SIZE}$. Again, we found this number to be curiously close to the Nvidia warp size. Further variations based on this procedure have indeed confirmed that one batch may reference no more than 32 unique vertices. Formulating it as a rule to apply to batch formation, we refer to the figure of maximum allowed unique vertices as $MAX\_UNIQUES$. We also noticed that occurrences of later ASR steps were shifted according to batches being terminated early due to the $MAX\_UNIQUES$ constraint. Hence, it follows that Nvidia GPUs can dynamically choose start and end positions for portions of the index buffer that qualify as a batch where reuse applies. In contrast, the identified boundary MAX_SIZE always applies on AMD GPUs without fail. There is no value for $MAX\_UNIQUES$ that causes irregularities in development of the ASR. For generality and notational convenience, we can, therefore, define that $MAX\_UNIQUES = MAX\_SIZE$ on AMD. For Intel, we

observed $MAX\_UNIQUES$ to equal 128, which has been reported on Haswell before (Barczak, 2016). For 128 different indices, $MAX\_SIZE = 1791$ can be observed. It seems that Intel's behavior can be described as MAX_SIZE being proportional to the number of different indices already being found, introducing batch boundaries depending on how many different indices have been recorded. A closer analysis of the hardware behavior also indicated that Intel internally completely sorts all indices and assigns them in alternating orders to 8-wide SIMD execution units.

Finally, we need to determine if and when the reusability of vertex information inside a batch can expire. We begin by considering the separating distance between indices as a potential factor. We start from our default repeating index buffer $\{0, 1, 2, 0, 1, 2, 0, \ldots\}$ of length MAX_SIZE and replace one entry at location i with a new index $X > 2$. Another instance of X will be placed at a distance n from the first, at position $i + n$. Increasing n towards the limit MAX_SIZE, a step in the ASR will then indicate that this particular vertex could not be reused over that distance. On Nvidia GPUs, we found that this is the case for $n = 41$. We noted a slight difference in the general retention behavior, depending on the API being used. On OpenGL, the identification of reusable vertices is always equivalent to a look-back of distance 42 into the index buffer. With DirectX, this behavior changes to a variable 42 ± 3 look-back, depending on the local index within a triangle, *i.e.*, the index buffer location modulo 3. Specifically, reuse is detected between two positions in the index buffer i, j ($i < j$) iff they reference the same vertex and their distance is d:

$$\begin{aligned} d \leq 44, &\quad \text{if } i \equiv 0 (\text{mod } 3) \\ d \leq 40 \vee d = 42, &\quad \text{if } i \equiv 1 (\text{mod } 3) \\ d \leq 42 \wedge d \neq 40, &\quad \text{if } i \equiv 2 (\text{mod } 3) \end{aligned}$$

In terms of data structures, the retention can thus be modeled as a FIFO cache of length $\sim$42 that always pushes queried entries, regardless of whether it already contains them or not. Any index that is not found in the retention model will count towards the number of unique vertices. On AMD, we could not identify a simple limiting separation distance within a batch. Instead, we tested whether the number of *unique* indices separating two instances of X would have an effect on the ASR. In fact, we found that reuse inside a batch on AMD does not work for two indices that are separated by more than 14 unique indices. Hence, the retention model of AMD for vertex reuse inside a batch can be expressed by a least-recently-used (LRU) cache of size 15. In contrast to Nvidia, this "forgetful" behavior does not influence

how batches are formed, however, it will obviously affect the ASR, since "forgotten" vertices that reoccur need to be shaded once more. On Intel, we could not identify a predictable condition for expiration of previously computed vertices above the already stated relationship between MAX_SIZE and $MAX\_UNIQUES$. Interestingly, we could observe certain vertices being shaded more often than others for no apparent reason, *e.g.*, for the repetitive sequence $\{0,1,2,0,1,2,\dots\}$, vertex 2 is shaded three times as often as 0 and 1, whereas for a sequence $\{0,1,2,3,0,1,2,3,\dots\}$ all vertices are shaded equally and overall fewer shader invocations occur than in the first case. Based on these observations, it seems impossible to capture Intel's retention model with a simple construct. Thus, assuming the best reuse scenario with sufficiently many different indices being rendered, we approximate Intel with a FIFO cache of size $MAX\_UNIQUES$.

Although we were able to identify these peculiarities in the architectures and verified them using the batch information produced according to Section 6.2.2, we can still observe unresolved artifacts in the batching behavior. For Nvidia, we found that the FIFO reuse does not hold if indices cross over a multiple of 2^{16}, *i.e.*, 16-bit boundary. For instance, if two indices I_a and I_b are placed in a batch where $\lfloor I_a/2^{16} \rfloor \neq \lfloor I_b/2^{16} \rfloor$, reuse according to the look-back retention may not occur. A potential explanation is an internal optimization for processing 16-bit wide indices, which can be handled more efficiently. For AMD, we found that the LRU size is also not always 15, but might sometimes be slightly longer or shorter. However, we were unable to find a comprehensive model to reproduce these exceptions. Analyzing the information of concurrently processed indices by wavefronts, we found that parts of different batches might be forwarded to the same wavefront on AMD, which ensures near perfect occupancy of the compute units. We assume that this combination can be modeled as a separate step after creating batches and identifying reuse within a batch with the sole goal of avoiding idle threads.

6.2.4. Predicting Batch Breakdown for the GPU

Given the information that we obtained by means described in Section 6.2.2 and interpreted in Section 6.2.3, we are able to make a qualified prediction about how a full list of triangle indices is split into separate batches on the GPU. Our base approach for predicting the batch breakdown is listed in Algorithm 2. Note that the properties that we identified for individual architectures can be abstracted by allowing the parameterization

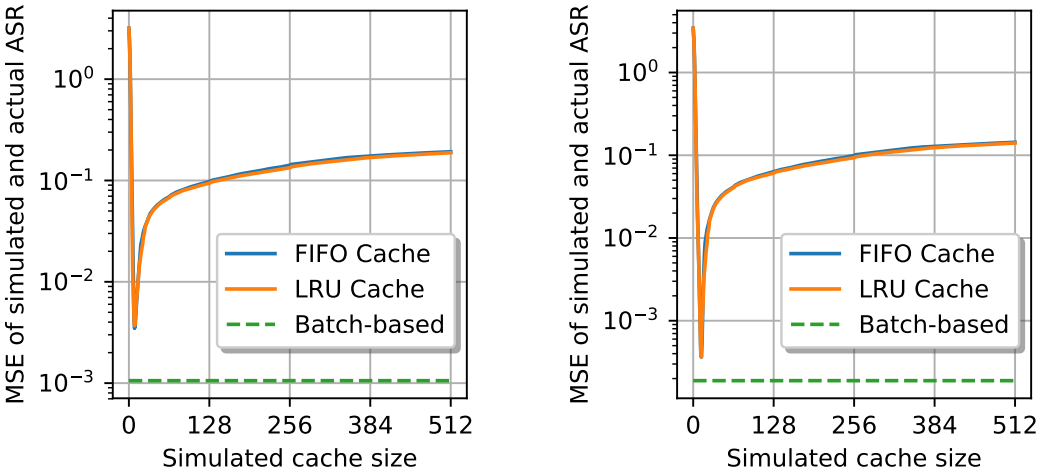
ALGORITHM 2: Predicting batch breakdown

```
1  in unsigned int Indices[]
2  out unsigned int invocations ← 0
   // Initialize
3  unsigned int pos ← 0
4  unsigned int size ← 0
5  unsigned int uniques ← 0
6  MemoryModule Available ← ∅
7  for each triangle △ in the index buffer Indices do
8      added ← 0
9      Available′ ← Available
10     for i ← 0 to 2 do
11         id ← ith index △(i) of triangle △
12         if {id} ⊄ Available′ then
13             added ← added + 1
14         else
15             Notify Available′ of id at (pos + i)
16         if (uniques + added) > MAX_UNIQUES or (size + i) > MAX_SIZE
            then
17             Found batch in Indices from (pos − size) to pos
18             Reset Available, initialize its contents to ∅
19             Available′ ← Available
20             size ← 0
21             uniques ← 0
               // Continue with current triangle △, load its i
                  indices into Available′
22             for j ← 0 to i do
23                 if △(j) ⊄ Available′ then
24                     Store △(j) in Available′
25                     added ← added +1
26                 else
27                     Notify Available′ of △(j) at (pos + j) .
28         else
29             Store id in Available′
30     Available ← Available′
31     pos ← pos + 3
32     size ← size + 3
33     uniques ← uniques + added
34     invocations ← invocations + added
```

of the *MemoryModule*, as well as the boundary values *MAX_SIZE* and *MAX_UNIQUES*. The *MemoryModule* encapsulates the behavior of how previously shaded data is maintained for quick reuse. *MAX_SIZE* and *MAX_UNIQUES* define the maximum allowed number of indices in a batch and of unique vertices used, respectively. The algorithm processes the index buffer in groups of three, assuming the input to be a triangle mesh. Shaded and available vertices (*uniques*) are tracked by the algorithm, as well as the total number of indices (*size*) in the current batch. If one of the boundary conditions is detected (line 16), we terminate the current batch and start a new one. This includes resetting the memory for available vertex information and—if a termination condition is detected mid-triangle—reloading its already visited indices (lines 22 to 27).

We evaluate the accuracy of our prediction scheme by simulating the batch breakdown for a number of commonly used 3D scenes (see Section 6.4). Nvidia's batching, as discerned above, can be achieved by setting *MAX_SIZE* to 96 and *MAX_UNIQUES* to 32. Availability of vertex data is tracked by the *MemoryModule*. For lack of a more accurate solution, we use a 15-wide LRU cache on AMD here and precise retention on Nvidia, according to the description in Section 6.2.3. As AMD shows no limitation of unique vertices for a batch, we set both *MAX_SIZE* and *MAX_UNIQUES* to the hard boundary of 384 indices. This effectively takes care of the first stage of AMD's two-tiered batching process. Although the second tier of AMD's approach influences the assignment and combination of vertices to wavefronts, it does not affect batching boundaries, nor the number of shader invocations. For the purpose of predicting or improving vertex reuse, the second stage can thus be ignored.

We keep the ASR returned by each simulation and compute the deviation from the actual measured ASR. We compare our results to global FIFO/LRU cache-based simulations of variable size. This is equivalent to the optimization scenarios used, *e.g.*, by Hoppe (1999) and Lin and Yu (2006), with look-ahead simulations to minimize the predicted ASR. Assessing the cache simulation accuracy at various sizes serves two relevant purposes: First, in order to irrefutably disprove existence of a centralized post-transform vertex cache, we want to ensure that no reasonably-sized cache can produce the ASR values we measured. Since we make no assumptions about the size of such a fictitious structure, we need to check a wide range of different sizes in order to conclusively rule out centralized caching. Second, it serves as an insight into which cache sizes actually come closest to the observed behavior. This information may guide researchers and developers to select the best parameters if they choose to continue using cache-based optimization algorithms.

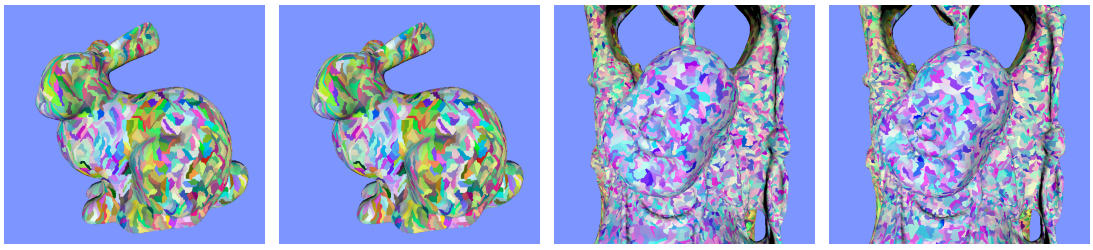


(a) Cache-based vs ours on Nvidia

(b) Cache-based vs ours on AMD (RX Vega 56)

Figure 6.2.: Accuracy of shading rate predictions using different simulation techniques. For Nvidia, the MSE with our batch-based approach is less than a third, for AMD it is half of the closest cache-based simulation. On Nvidia, the minimum occurs at a cache size of 10 for FIFO, on AMD at 15 for LRU. Although based on a single centralized cache instead of multiple parallel ones, these results seem to reflect our assumptions for retention on the respective architectures.

The mean squared error (MSE) to the actual measured ASR on the GPU (averaged over our full test suite) is shown in Figures 6.2a and 6.2b. Evidently, our batch-based approach is significantly more accurate than cache-based simulations. At their best configuration (*i.e.*, with a cache size of 10/15), the MSE of FIFO and LRU variants is still more than $3\times$ that of ours on Nvidia, and $2\times$ on AMD. In fact, our prediction for batches and ASR on Nvidia are completely accurate for models that contain fewer than 2^{16} indices. The remaining reported error stems from our inability to predict batching of scenes that are bigger than that. As mentioned above, Nvidia appears to employ an (as of yet unidentified) mechanism that avoids reuse of indices in the same batch if their integer division by 2^{16} yields different results. We visualize the adverse effect of this property on the quality of our batch prediction algorithm in Figure 6.3. On AMD, the slightly higher residual error is due to behavior not fully captured by our batching function, including the exact size for LRU retention, as discussed in Section 6.2.3.



(a) Bunny prediction (b) Bunny measured (c) Buddha prediction (d) Buddha measured

Figure 6.3.: Rendering of the predicted and actually reported batches as they are processed on Nvidia for the `bunny` and `happy_buddha` models. Random colors are used to distinguish batch boundaries. For scenes with a vertex count $< 2^{16}$, such as `bunny`, we can perfectly predict how batches are formed (a, b). For larger scenes, our model is not exact, but can still predict ASR and overall batch generation more accurately than cache-based simulations (c, d).

6.3. Batch-based Mesh Optimization

In order to determine whether knowledge of batch boundaries can be beneficial for preprocessing algorithms, we design a simple and concise mesh reordering algorithm based on the knowledge we gathered from observed hardware behavior. Focusing on low runtime, our algorithm follows a greedy strategy that continuously grows batches by choosing an origin and iteratively adding faces to the batch. Similar to existing approaches, the selection of the next face to be added to the mesh in each step is governed by a cost or priority function that identifies the most suitable candidate. The full algorithm and its integration of the batch prediction function are outlined in Algorithm 3.

Our algorithm considers four parts for the priority of each face: *Vertex Reuse* R, *Vertex Valence* V, *Face Distance* D and *Batch Neighborhood* N. Priority p for triangle $\triangle$ is computed by weighting the factors with appropriate coefficients:

$$p(\triangle) = k_r \cdot R(\triangle) + k_v \cdot V(\triangle) + k_d \cdot D(\triangle) + k_n \cdot N(\triangle). \tag{6.1}$$

- *Vertex Reuse* captures the number of vertices of a triangle that will result in a reuse of the vertex shading information when the triangle is added to the current batch, *i.e.*, how many of its vertices are already in the batch and will be identified for reuse when this triangle is added. Obviously, choosing a high priority k_r is important to build batches that result in as few vertex shader invocations as possible. To determine how many vertices of a triangle actually will be reused, it is essential to know how and when the hardware introduces batch boundaries and how it identifies reusable vertex information.

- *Vertex Valence* corresponds to the sum over all vertex valences of the triangle, whereby whenever a triangle is added to the index buffer, the valence for all its vertices is reduced by one. One can think of this as making a copy M_{copy} of the input mesh on which the vertex valence is tracked and whenever a triangle is added to the index buffer, it is removed from the mesh copy. Using a negative priority k_v will prioritize triangles with low valence, *i.e.*, those which will likely have a low chance of leading to reuse if they are left over. This approach also guarantees that the algorithm will start batches from mesh boundaries (and from boundaries of previous batches) and not randomly grow isolated batches all over the mesh.
- *Face Distance* is computed on the dual graph of the mesh and corresponds to the sum over all distances from all faces currently in the batch to all surrounding faces, *i.e.*, triangles that can be reached from all triangles in the batch over few other triangles show a small D. This means that a negative k_d will prioritize those triangles that can be reached easily, and will thus lead to more 'circular' batches. On the other hand, a positive k_d will lead to elongated batches. Especially under the assumption that vertex reuse is possible among all triangles within a batch, 'circular' batches are preferable as they will show the highest potential reuse.
- *Batch Neighborhood* is 1 for triangles that are direct neighbors to triangles which have been added to completed batches, *i.e.*, are part of the boundary of M_{copy} but are not part of the boundary in the original mesh. Thus, N allows to guide batches along already existing batches (positive k_n) or slightly push batches away from already existing ones (negative k_n). The former helps to avoid fragmentation of batches; the latter helps to avoid elongated batches, especially when $k_d = 0$.

In order to increase the probability of a cache hit, previous work usually considers the immediate neighborhood of the last processed triangle first, thereby catering to the presumed, underlying cache architecture. In contrast, we argue that the order of faces within a batch is irrelevant for mesh optimization, as long as the hardware can identify that the same vertices are referenced. Furthermore, knowing when the hardware inserts a batch boundary, allows an optimization algorithm additional freedom, as there is no possibility for the next triangle to create any reuse with previously referenced vertices. Thus, the optimization algorithm can 'jump' to any other mesh location. Especially for a hardware that inserts batch boundaries often, *e.g.*, Nvidia, these jumps have a significant impact on the overall performance. In the mindset of previous algorithms, they can be seen as very frequent cache resets.

ALGORITHM 3: Batch-based Optimization with Batching Predictor bp

```
1  in VertexInformation vI[]
2  in unsigned int Indices_orig[]
3  out unsigned int Indices_opt[]
4  Set reuseVertices ← ∅
5  Map distance ← ∅
6  PriorityQueue queue ← ∅
7  for each triangle △ ∈ Indices_orig do
8      V(△) ← Σ_{i=0}^{2} vI[△(i)].valence
9      Insert △ with score V(△) · k_v into queue
10 while queue ≠ ∅ do
11     △ ← highest-scoring entry in queue
12     if can add △ according to bp then
13         Pop △ from queue and add it to Indices_opt
           // add R(·) and reduce V(·), add D(·) according to new △
14         for i ← 0 to 2 do
15             r_i ← 0 if △'(i) ∈ reuseVertices else 1
16             for △' ∈ vI[△(i)].faces do
17                 adjust p(△') by (r_i · k_r − k_v)
18         for △' ∈ neighbors of △ do
19             adjust p(△') by k_d · D(△, △')
20             distance(△') ← k_d · D(△, △')
21         Add indices of △ to reuseVertices
           // update R(·) for 'forgotten' vertices
22         for v ∈ reuseVertices do
23             if v can no longer be reused according to bp then
24                 remove v from reuseVertices
25                 for △' ∈ vI[v].faces do
26                     adjust p(△') by −k_r
27     else
           // remove R(·) due to batch end, remove D(·) and add N(·)
28         for v ∈ reuseVertices do
29             for △' ∈ vI[v].faces do
30                 adjust p(△') by −k_r
31         for △' ∈ neighbors of △ do
32             adjust p(△') by k_n − distance(△')
33         Reset bp for new batch
34         reuseVertices ← ∅
35         distance ← ∅
```

Computing and updating these factors for all triangles whenever a triangle is added to the current batch or a batch is completed, continuously changes the priority of all potential triangles. Using a priority queue with low update complexity when the priorities are increased or decreased is essential for efficient processing. To this end, we employ a Fibonacci Heap as our priority queue. Empiric assessment has shown that the configuration

$$k_c = 1024, k_v = -4, k_d = -1, k_n = -1 \tag{6.2}$$

yields consistently good results. In terms of computational effort, distance information D is by far the most expensive factor of our cost function, since its maintenance requires updates to all faces that are neighbors to the current batch. In order to reduce processing time while maintaining low ASR, we can eliminate D from the cost function and set $k_v = k_n = -3$.

The previously discussed algorithm is based on some simplified views on the hardware. As mentioned above, we have found that indices that go across multiples of 2^{16} on Nvidia will prohibit reuse within a batch. Thus, we added a simple Nvidia post-processing step that operates on the preprocessed mesh, to make sure that the simulated batch boundaries are not destroyed by this behavior. Whenever we determine (based on our simulation) that a batch contains indices that cross a 16-bit boundary, we duplicate conflicting vertices and add them to the end of the vertex buffer. Obviously, this can again result in the batch crossing over 16-bit boundaries (if the previously largest index is not in the same 16-bit boundary as the new vertex buffer size), which may require us to copy another set of vertices to the back. Instead of referencing the original lower vertices, the indices are then recast to reference the newly added ones. Note that this step increases the total vertex count and reduces the *ideal* reuse potential of the mesh. However, it results in better *achieved* reuse within batches and improves ASR on Nvidia, as we will shortly see.

6.4. Evaluation

To compare with previous work, we use readily available libraries for algorithms by Hoppe (1999) and Sander, Nehab and Barczak (2007) from *DirectXMesh* and *Tootle*, respectively. For the work by Forsyth (2006), we used the standalone version (TomF), provided by Adrian Stone, as referenced in the original publication. Regarding K-Cache optimization (Lin and Yu, 2006), we consider our own, faithful C/C++ implementation.

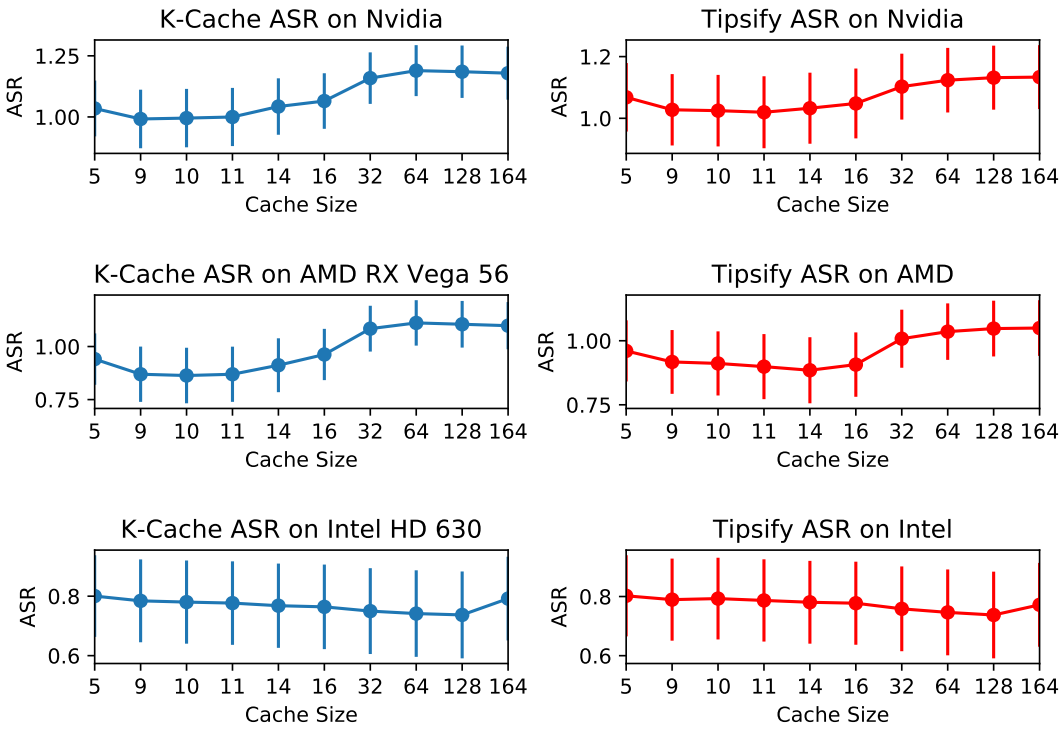


Figure 6.4.: We evaluate the ASR of meshes processed with differently parameterized optimization methods K-Cache and Tipsify on Nvidia, AMD and Intel hardware. Similar to our prediction models, Nvidia and AMD achieve best performance with cache-based optimizations at sizes between 10 and 16. On Intel, best reuse occurs at a cache size of 128 which is the maximum number of reusable vertices.

For input data, we include a variety of commonly used test scenes from the graphics community, as well as clip-space geometry that we captured from five recent video games and an Nvidia technical demo: Age of Mythology (abbreviated am), Assassin's Creed: Black Flag (as), Deus Ex: Human Revolution (dx), Stone Giant animation (sg), Total War: Shogun 2 (sh), Rise of the Tomb Raider (tr), and The Witcher 3 (tw). We process each input scene with the above algorithms and record the achieved ASR when rendering them on different GPU models, as described in Section 6.2.1. To enable a fair comparison, we selected the cache sizes for K-Cache and Tipsify that produced the best results (see Figure 6.4). Hence, these results serve not only as a baseline to compare against but also as a survey on the potential performance of previous approaches on modern hardware. All models were rendered as indexed triangle lists using 32-bit indices. For smaller models, we also tested 16-bit indices and found that it does not influence the resulting ASR at all.

The lowest shading rate with K-Cache was achieved with a cache size of 10 entries on Nvidia and AMD. For Tipsify, we used a cache size of 11 on Nvidia, and 14 on AMD. This is in accordance with our previous assessment regarding cache-based ASR prediction: given that these algorithms simulate and optimize the ASR based on the existence of a cache, they perform best at the configuration where a cache-based prediction is closest to the actual behavior, *i.e.*, at a cache size between 10 and 16. On Intel, as previously assumed, the best cache size equals its $MAX\_UNIQUES$ figure (128).

For Nvidia GPUs, we tested all techniques on the Geforce GTX 780Ti, 980Ti, 1060, 1080 and 1080Ti, covering three different hardware generations (Kepler, Maxwell and Pascal). Recorded shading rates were identical in all cases, regardless of the particular model used, and are shown in Table 6.1. For reference, we also give the ideal ASR values, *i.e.*, shading rates that would be achieved with absolute reuse of shaded vertex output. As documented by these results, our simple, batch-based algorithm yields the lowest shading rate out of all contestants in every test scenario but one. For large, carefully crafted meshes, the benefit of our algorithm over the best contender is significant: for the Happy Buddha statue, we report a 15% improvement over the best alternative, K-Cache. For the dragon model, this figure jumps to a 25% improvement over the best result yielded from previous work by Hoppe. For weakly structured meshes, such as the tree model and scenes obtained from captured video game frames, the improvement of the ASR is smaller, but usually still significant in the 3-10% range.

A large portion of this improvement is hinged on our post-processing step, which makes sure indices within a batch will not cross 16-bit boundaries. As shown above, we cannot exactly predict Nvidia's batching for arbitrary input models that contain more than 2^{16} unique vertices (see Figure 6.3). Note that ignoring this property would mean that the algorithm assumes reuse within batches that do not align with those constructed by the GPU, and virtually all potential benefit is annulled. For game scenes, optimizing without this post-processing step usually results in a 1-10% increase of the ASR. In contrast, for `happy_buddha` and `xyz_dragon`, ignoring 16-bit boundary constraints yields a much higher ASR of 1.01 and 1.15, respectively.

For AMD, we considered an R7 200 from the GCN2 generation, as well as an RX Vega 56. For the AMD models, the results of batching were much less rewarding than for Nvidia. Although our batch-based algorithm is on par with more elaborate techniques, it is usually bested by either K-Cache and TomF. We note here once more that the identification of the AMD batching functions

Table 6.1.: Achieved ASR with different mesh optimization techniques applied, on recent Nvidia generations. For all cases but one, our simple, batch-based mesh optimization can outperform elaborate methods, most prominently for the `happy_buddha` and `xyz_dragon` test cases. For reference, we also include the ASR that would be possible with perfect reuse (*e.g.*, using a centralized cache of infinite size).

	Hoppe	K-Cache	Tipsify	TomF	Ours	*Perfect Reuse*
sphere	0.83	0.82	0.83	0.88	**0.81**	*0.50*
bunny	0.84	0.84	0.86	0.88	**0.82**	*0.50*
happy_buddha	0.98	0.95	0.98	0.99	**0.81**	*0.50*
xyz_dragon	1.07	1.10	1.08	1.16	**0.82**	*0.50*
tree	2.07	2.09	2.09	2.08	**2.06**	*2.06*
am01	0.97	0.86	0.88	0.87	**0.84**	*0.60*
am02	0.95	0.81	0.81	0.81	**0.78**	*0.48*
as01	0.87	0.85	0.88	0.86	**0.83**	*0.59*
as02	1.27	1.26	1.28	1.26	**1.24**	*1.11*
dx01	0.88	**0.85**	0.90	0.85	0.89	*0.61*
dx02	0.87	0.84	0.88	0.85	**0.84**	*0.62*
sg01	0.87	0.83	0.88	0.84	**0.83**	*0.53*
sg02	0.89	0.85	0.89	0.85	**0.84**	*0.56*
sh01	1.01	0.97	1.00	0.97	**0.92**	*0.74*
sh02	0.98	0.95	0.98	0.95	**0.94**	*0.74*
tr01	0.95	0.89	0.93	0.89	**0.87**	*0.68*
tr02	0.93	0.89	0.92	0.89	**0.88**	*0.66*
tw01	0.87	0.87	0.89	0.89	**0.84**	*0.55*
tw02	1.43	1.39	1.41	1.39	**1.37**	*1.23*

is likely incomplete. However, the properties of the AMD architecture also reduce the impact that correct batching can have. Given the combination of large batches and small cache size, elaborate algorithms focusing on cache optimization achieve ASR rates that are already close to ideal. While fully understanding and incorporating knowledge about batch boundaries into a sophisticated algorithm is guaranteed to yield better reuse, our simple optimization algorithm mostly relies on ASR improvement through approximate batching, which is insufficient for AMD GPU models. Recorded ASR values are listed in Table 6.2a.

Table 6.2.: ASR for all test scenes with optimization techniques applied. We provide recorded results for AMD RX Vega 56 (virtually identical to those by the R7 200) and Intel HD 630. Although the performance of our simple batch-based method is usually on par with more sophisticated techniques, it is commonly bested by either the K-Cache or Tipsify algorithms.

	(a) AMD					(b) Intel				
	Hoppe	K-Cache	Tipsify	TomF	Ours	Hoppe	K-Cache	Tipsify	TomF	Ours
sph	**0.66**	0.67	0.68	0.77	0.72	0.59	0.58	**0.52**	0.58	0.60
bun	**0.68**	0.70	0.72	0.77	0.72	0.60	0.53	**0.53**	0.57	0.58
bud	0.73	**0.71**	0.75	0.74	0.75	0.61	**0.55**	0.55	0.55	0.62
dra	**0.67**	0.69	0.71	0.77	0.72	0.60	0.53	**0.52**	0.57	0.60
tree	2.06	2.07	2.07	2.06	**2.06**	2.06	2.06	2.06	2.06	**2.06**
am1	0.85	**0.74**	0.76	0.77	0.77	0.72	**0.60**	0.60	0.65	0.69
am2	0.81	0.68	0.68	0.69	**0.68**	0.67	**0.48**	0.48	0.55	0.60
as1	0.74	**0.73**	0.75	0.74	0.75	0.62	**0.59**	0.60	0.60	0.64
as2	1.19	**1.19**	1.20	1.19	1.20	1.14	**1.11**	1.12	1.12	1.14
dx1	0.77	**0.75**	0.79	0.75	0.82	0.64	0.63	0.65	**0.62**	0.73
dx2	0.75	**0.73**	0.76	0.74	0.74	0.63	**0.62**	0.62	0.62	0.65
sg1	0.73	**0.71**	0.75	0.73	0.74	0.57	**0.54**	0.55	0.55	0.60
sg2	0.77	**0.73**	0.77	0.75	0.76	0.61	**0.56**	0.57	0.58	0.63
sh1	0.88	**0.84**	0.86	0.84	0.84	0.79	**0.74**	0.74	0.75	0.77
sh2	0.87	**0.85**	0.88	0.86	0.86	0.77	**0.74**	0.75	0.77	0.78
tr1	0.83	**0.78**	0.81	0.79	0.80	0.72	**0.68**	0.68	0.69	0.71
tr2	0.81	**0.78**	0.81	0.79	0.79	0.70	**0.66**	0.67	0.67	0.70
tw1	**0.72**	0.73	0.75	0.78	0.75	0.63	**0.57**	0.57	0.60	0.63
tw2	1.35	**1.31**	1.33	1.32	1.32	1.29	**1.23**	1.24	1.24	1.27

Similar trends can be observed for our evaluation on Intel GPUs (Table 6.2b). Since their architecture features the biggest batch size and the largest cache, the achieved ASR values with cache-based algorithms are even closer to ideal reuse rates. Additionally, as previously mentioned, batching alone does not suffice to fully explain the formation of the shading rate on Intel. Hence, our simple batch-focused algorithm is on par with Hoppe's optimization, but newer, cache-focused algorithms provide a better choice overall.

6.5. Discussion

Although modern GPU hardware evolves at a staggering speed, the existence of a central, post-transform vertex cache is still considered to be self-evident by many. Our experiments conclusively show that this assumption does no longer match the behavior that we observe on these massively parallel devices. The most reasonable and, in fact, most likely alternative is the batch-driven decomposition of input vertex data so that it can be efficiently handled in separate, independent shading units to achieve adequate load balancing for the vertex processing portion of the pipeline. By drawing a strong association between the hardware design and information obtained through indirect measurements, we have shown that it is possible to predict this decomposition process with high accuracy. Understanding the subdivision and distribution of vertex input on GPUs has several important implications for research into mesh optimization algorithms: previous work commonly uses cache-based software simulations to illustrate the merit of their work. However, during our evaluation, we have shown that the actually achieved vertex shading rates differ significantly from those reported by such simulations.

Knowledge of the GPU batch functions provides us with a reliable tool for off-line simulation and evaluation of mesh optimization algorithms. Given the exact parameters of the batch function, we were able to devise a preprocessing algorithm to improve the vertex shading rate on modern GPUs beyond the capabilities of previous work. Furthermore, since hardware simulation and state prediction is oftentimes included in the optimization algorithm itself, our work opens the door for the development of new algorithms that may achieve even lower shading rates and better vertex reuse on contemporary GPU hardware. We have made our source code for reuse analysis and optimization available to the public at `https://github.com/GPUPeople/vertex_batch_optimization`. Lastly, challenging a concept as established as the post-transform cache and finding a more precise fit raises the question what other pipeline stages, assumptions and conventional notions on GPU rendering should be revisited.

7. Binning Patterns for Balanced Sort-Middle Rendering

Contents

Having observed the policies of modern GPUs to consolidate both vertex reuse and effective load balancing in a massively parallel environment, we move to the rasterization stage and examine the assignment of primitives to framebuffer tiles. Again, we assume that the implementation of this pipeline stage in hardware is governed by considerations for load balancing that we aim to identify, enhance and apply in our parallel software renderer, CURE.

7.1. Sort-Middle Rendering & Load Balancing

Work distribution strategies for parallel rendering have been classified by Molnar, Cox et al. (1994) based on where in the pipeline redistribution between processors occurs: before geometry processing (sort-first), between geometry and fragment processing (sort-middle), or after fragment processing (sort-last). In a sort-middle approach, the output domain is spatially subdivided into multiple *bins*. Primitives that are generated by the geometry stage are sorted into the bins they overlap before rasterization and fragment shading occur. Each bin is then assigned to one or several processing units, ideally in such a way that leverages near-uniform load for all participating processors.

Sort-middle seems to be the preferable strategy on modern GPU hardware, since the screen area covered by output primitives is easy to compute after geometry processing and the transfer overhead is relatively small compared to after they have been split into a large number of fragments. GPU rasterization typically uses a coarse-to-fine strategy and is implemented on special-purpose hardware units called *rasterizers*. The assignment of bins to rasterizers commonly follows a static, fixed spatial pattern. Using such a static pattern for load balancing seems straightforward, but also rather well-reasoned, given that it grants each rasterizer exclusive access to the frame buffer in the assigned region, which eliminates the problem of resource contention and has a positive impact on cache performance.

It stands to reason that the design of such a binning pattern can have an appreciable effect on sort-middle rendering performance. If the distribution of fragments is not sufficiently uniform across screen space, the resulting load imbalances will turn individual rasterizers to bottlenecks. If we reject the option of dynamic load balancing because of its elevated hardware demands, using a suitable static binning pattern seems a requirement for achieving adequate rasterization performance on GPUs. Furthermore, as the trend of increasing processor counts continues, good scalability of binning patterns may become relevant in future hardware. Thus, we found it surprising that detailed advice on patterns with good load balancing characteristics is scarce in related literature. With screen resolutions reaching 4K or even 8K and parallel rendering ranging from smartphones to desktop systems and even into the cloud, load balancing strategies become more and more important.

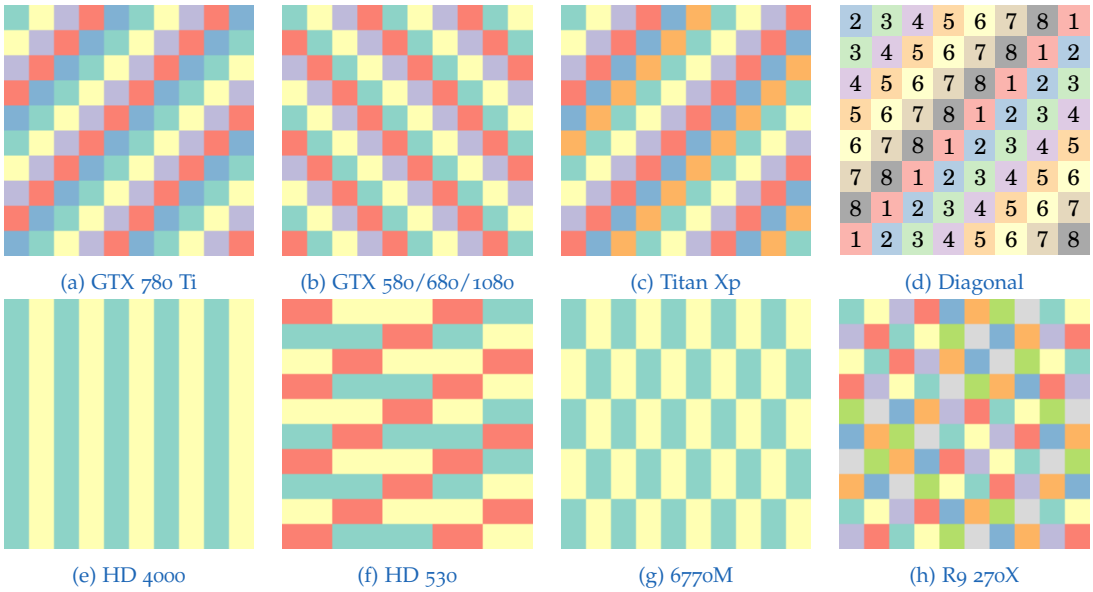
In order for static binning to be globally applicable, patterns must be scalable with respect to bin size, number of rasterizers, and screen size. In this chapter, we investigate these issues in detail and make the following contributions:

1. We determine and analyze the patterns employed on the GPU throughout recent years.
2. We analyze real-world rendering workloads from recent video games and derive requirements for effective patterns.
3. We present ten different pattern design strategies based on these requirements and previous work.
4. We assess and compare the load balancing characteristics of the proposed patterns on more than 200 game scenes.
5. We evaluate the effects of the most promising patterns on overall performance in CURE, a GPU software rendering pipeline.

7.2. Built-In GPU Patterns

On the GPU, physical cores may be grouped together hierarchically to form powerful logical processing clusters. In Nvidia architectures (NVIDIA, 2009), up to 30 streaming multiprocessors (SMs), each capable of maintaining thousands of threads, are grouped into a small number of graphics processing clusters (GPCs). Workload distribution for rasterization occurs only on GPC level of the architecture, with each GPC representing a logical rasterizer.

To analyze binning patterns on Nvidia GPUs, we use a custom GLSL shader and the `NV_shader_thread_group` extension to identify pixel locations that submit to the same group of SMs and are therefore handled by the same GPC. Based on these experiments and the architecture specification, we have recreated the patterns that are used for workload distribution during hardware rasterization. Figure 7.1 shows these patterns for multiple hardware generations. According to our results, on five out of six recent flagship models (Figures 7.1a–c), a diagonally aligned pattern at 45° is used for identifying responsible rasterizers. A corresponding assignment map for 8 rasterizers is shown in Figure 7.1d. While we also found less obvious patterns (particularly in models with imbalanced GPCs), these violate our defined requirement for patterns being scalable, as they cater to one precise configuration and could not be ported to another model. However, even for them, a trend of diagonal alignment of bins is prominent. Regardless of the patterns used, bin sizes are consistently small at 16×16 pixels in an effort to mitigate load imbalance (Purcell, 2010). We also consider Intel and AMD models (Figures 7.1e–h), for which we used a timing-based approach to identify screen regions that influence each other's performance for processing fragments. On those architectures, load balancing appears to not be limited to rasterizers alone.



(a) GTX 780 Ti (b) GTX 580/680/1080 (c) Titan Xp (d) Diagonal

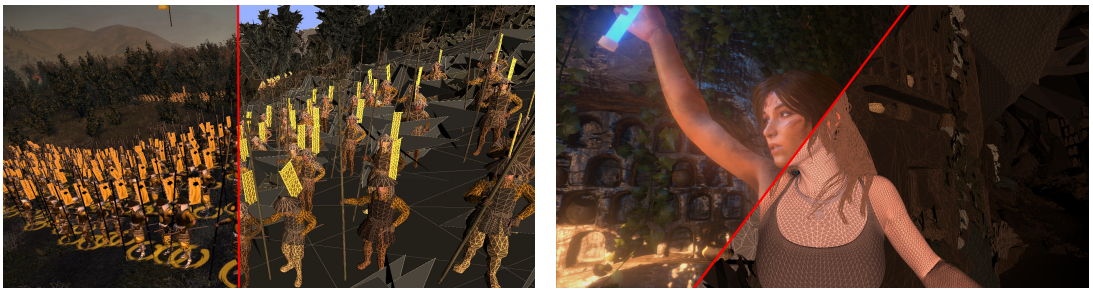
(e) HD 4000 (f) HD 530 (g) 6770M (h) R9 270X

Figure 7.1.: Observed patterns for workload distribution used in GPUs by Nvidia (a–c), which seem to follow a diagonal (d) pattern. (e, f) Patterns on Intel and (g, h) AMD.

For the AMD R9 270X and HD 7870 models, *e.g.*, we detected 8 separate domains distinctly sharing rendered workload, in contrast to the 2 officially documented rasterizers available (AMD, 2012). We suspect that this is due to those architectures employing a more fine-grained load balancing concept directly between individual compute units. The full list of detected patterns and description of timing-based experiments are available in Appendix B.

7.3. Guidelines for Pattern Designs

An effective static bin pattern for load balancing in rasterization should consider the workload characteristics of typical rendering content. To this end, we identify fundamental caveats and run statistical tests on versatile, authentic computer graphics scenes to derive guidelines for pattern design. To obtain a manageable parameter space, we adopt the following restrictions regarding layout combinations for bins and viewports: First, we only consider square bin resolutions, which is common practice in existing software and hardware pipelines. Second, bin sizes are considered to be an immutable property of the underlying architecture. Third, the pattern layout must not depend on high-level parameters, such as window resolution or size. Fourth, all analytical



(a) Total War: Shogun 2 (2011)

(b) Rise of the Tomb Raider (2015)

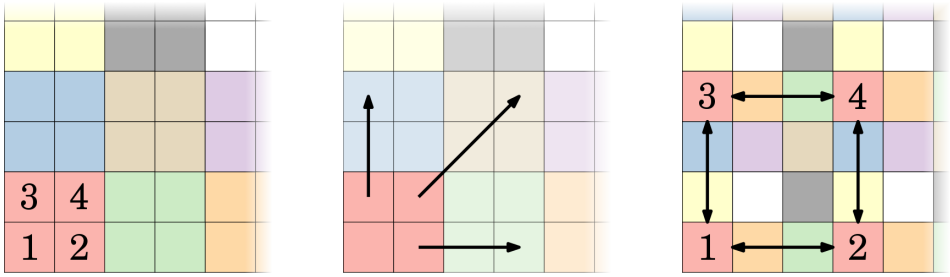
Figure 7.2.: Selected game scenes from our dataset. Images show the original rendering with an overlay of the output by the software rendering pipeline we used to verify our analytical results. Not shown here: *Deus Ex: Human Revolution* (2011), *Tomb Raider* (2013), *Assassin's Creed IV: Black Flag* (2013), *Age of Mythology* (2014), *The Witcher 3: Wild Hunt* (2015) and Nvidia's *Stone Giant* technical DirectX 11 demo. **Total War: Shogun 2 capture courtesy of The Creative Assembly. Rise of the Tomb Raider screenshot courtesy of Crystal Dynamics.**

and practical evaluations focus on applications running in landscape mode at screen resolutions with a 16:9 aspect ratio. In the following sections, we assume that a low variance in fragment load across all rasterizers provides a meaningful indicator of effective workload balancing and draw conclusions for suitable design choices accordingly. Once established, we assess and compare the relative benefits of individual patterns in the context of software rasterization with these theoretical assumptions.

Our guidelines are grounded in both a theoretical analysis and in an in-depth evaluation of a representative dataset modeling typical GPU workloads. In order to faithfully reproduce realistic GPU rasterization tasks, we have selected seven video games and a recent Nvidia tech demo (see Figure 7.2). For each of these applications, we have captured the triangle stream for at least 20 frame snapshots by injecting a custom DirectX 11 DLL, which saves clip-space data directly to a file. All capturing was done at 1080p resolution.

7.3.1. Space Utilization

An efficient binning pattern must respect the number of available rasterizers in its layout. Consider a naïve binning policy for N parallel rasterizers, where each rasterizer is assigned to an entire row of bins on the viewport (*cf.* Crockett and Orloff, 1993). Although this policy may achieve satisfactory results with a small bin height h and low number of rasterizers N, choosing either value



(a) Initial state (b) First iteration (c) New layout

Figure 7.3.: Progression for spacing out clustered bins.

such that $h \cdot (N-1)$ surpasses the vertical resolution of the viewport implies that at least one rasterizer remains idle. Such a naïve pattern would, therefore, not scale well with increasing rasterizer count. Given the restriction that bin sizes are immutable, the only option towards a scalable binning policy is to pack bins for all available rasterizers into as small a 2D region as possible and thereby increase the likelihood of rasterizer participation.

7.3.2. Local Clustering of Geometry

In most scenes, geometry is not uniformly distributed (M. F. Deering, 1993). Decorative elements such as grass, soil or water are represented with low geometric density, while prominent objects, such as trees, edifices or living entities, are designed with a stronger emphasis on geometric detail. These observations suggest that the geometry of a 3D scene, projected to a 2D viewport, has a tendency to form local clusters in screen space. In this case, assigning one rasterizer to contiguous bins could considerably hurt performance.

In order to quantify the influence of local geometry clusters on load balancing, we have assessed the potential improvement that can be achieved by iteratively increasing the distance between bins assigned to the same rasterizer, as outlined in Figure 7.3. For our evaluation running at 1080p, we first assume a bin size $X \times X$ and subdivide the screen into quad regions (Figure 7.3a) containing four bins each. Each quad is then assigned its own dedicated rasterizer, and we compute the reference standard deviation σ_{ref} over all rasterizers from the ideal fragment load in this configuration. We then proceed to break up the quads (Figure 7.3b) by incrementally increasing the distance between the bins assigned to each rasterizer. From each original quad, one bin location is moved two slots to the right, one is moved two slots upward and

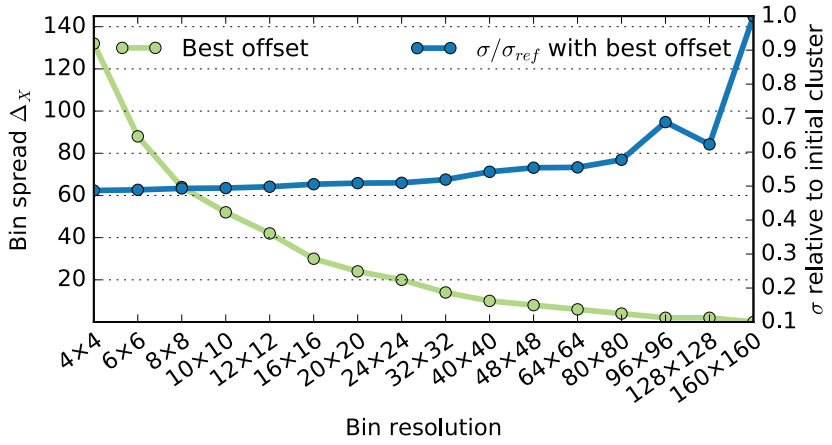


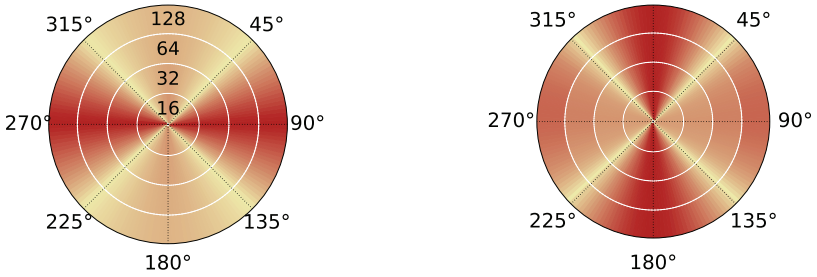
Figure 7.4.: Spreading out bins assigned to the same rasterizer lessens the impact of triangle clusters and reduces load variance. Smallest σ occurs at farthest separation.

one is moved both upwards and to the right. After the first iteration, the gap between bins assigned to the same rasterizer spans two slots (Figure 7.3c). We continue to space out the bin locations in this way until the distance between the bins for each rasterizer surpasses half of the viewport height. In each iteration i, we record the current standard deviation σ_i of rasterizer workload and compare it to σ_{ref}. The ideal distance between rasterizers is given by the iteration index i that produced the smallest ratio σ/σ_{ref}, multiplied by two.

Figure 7.4 shows the results of this experiment for several different bin sizes, averaged over our entire dataset. Note that the best offset for a bin size $X \times X$ is usually found at or close to distance Δ_X such that $2 \cdot \Delta_X \cdot X = 1080$. Thus, the ideal distance between bins assigned to the same rasterizer is equal to their maximal possible separation. Furthermore, the effect of breaking up the original clusters has a major impact on the variance, reducing σ by at least 30%. Note that, at 160×160, no offset occurred, since $160 \cdot 4 > \frac{1080}{2}$.

7.3.3. Influence of Orientation

Both space utilization and local clustering behavior imply that bins assigned to the same rasterizer should be as widely spaced out as possible in order to avoid local rasterizer repetition. However, we have yet to establish whether the impact of these repetitions is equally severe in all directions. In real-world scenarios, gravity ensures a natural preference of horizontal structures in bedrock, bodies of water and terrain. In contrast, man-made structures,



(a) Full Test Set

(b) Age of Mythology Only

Figure 7.5.: Average variance of fragment load when dividing the viewport into pixel lines with different orientations. Numbers inside the left circle give the line length N that is illustrated by the corresponding ring. Horizontal and vertical directions exhibit high variance (red). Only in *Age of Mythology* is the variance higher with vertical lines than with horizontal.

plants and animate entities often stand upright, affording them a superior vantage point. As a matter of fact, very few elements remain that are naturally diagonally aligned. Graphics applications aiming to present realistic scenes exhibit similar structural properties in their geometry. Consequently, we can assume that horizontal and vertical rasterizer repetitions make a pattern more susceptible to localized triangle clusters and therefore more likely to suffer from workload imbalances.

In order to confirm this theory, we analyze the influence on load variance when subdividing the viewport into screen space lines of varying orientation. Samples for computing variance are taken at pixel level by setting up a sliding window of $N \times N$ pixels and sampling N consecutive pixel locations along a line through the center of the sliding window in a given direction, defined by an angular parameter. Line sampling is performed using a Bresenham algorithm aligned with the desired angle. The sum of the fragments submitted to the N pixels in each line is recorded. We then compute the standard deviation σ of the fragment counts for the given direction over all sliding window positions. σ is further normalized for each scene by dividing with the average number of fragments per pixel. This process is performed for a discrete set of directions, yielding the average variance for each tested orientation. We list the extrema of normalized average standard deviations, as well as the corresponding orientations for all tested applications and sliding window resolutions in Table 7.1. Furthermore, Figure 7.5 provides a more intuitive classification of all tested directions on our dataset, with each concentric circle representing a different line length N.

Table 7.1.: Direction of the lowest and highest average standard deviation (normalized) along all directions for different applications. High σ indicates that an effective pattern should avoid assigning bins along those directions to the same rasterizer.

N	AoM		AC 4		TR		RTR		SG		SH 2		DX: HR		TW 3	
	σ_{lo}	σ_{hi}	σ_{lo}	σ_{hi}	σ_{lo}	σ_{hi}	σ_{lo}	σ_{hi}	σ_{lo}	σ_{hi}	σ_{lo}	σ_{hi}	σ_{lo}	σ_{hi}	σ_{lo}	σ_{hi}
16	1	1.04	1	1.03	0.64	0.65	0.96	0.98	0.52	0.53	1.61	1.7	0.7	0.72	1.5	1.56
	(46°)	(0°)	(44°)	(90°)	(44°)	(90°)	(44°)	(94°)	(44°)	(90°)	(44°)	(92°)	(44°)	(90°)	(46°)	(92°)
32	0.95	1	0.96	1.01	0.61	0.63	0.92	0.96	0.51	0.52	1.51	1.67	0.68	0.71	1.4	1.53
	(46°)	(0°)	(44°)	(90°)	(44°)	(90°)	(44°)	(90°)	(46°)	(90°)	(136°)	(90°)	(44°)	(90°)	(136°)	(90°)
64	0.88	0.94	0.92	0.99	0.6	0.61	0.86	0.91	0.5	0.51	1.39	1.63	0.65	0.68	1.29	1.48
	(44°)	(0°)	(44°)	(90°)	(46°)	(90°)	(44°)	(88°)	(44°)	(90°)	(44°)	(90°)	(44°)	(90°)	(136°)	(90°)
128	0.78	0.85	0.84	0.95	0.56	0.59	0.78	0.85	0.47	0.5	1.22	1.58	0.60	0.66	1.15	1.42
	(136°)	(0°)	(44°)	(90°)	(44°)	(90°)	(44°)	(88°)	(44°)	(90°)	(44°)	(90°)	(44°)	(90°)	(46°)	(90°)

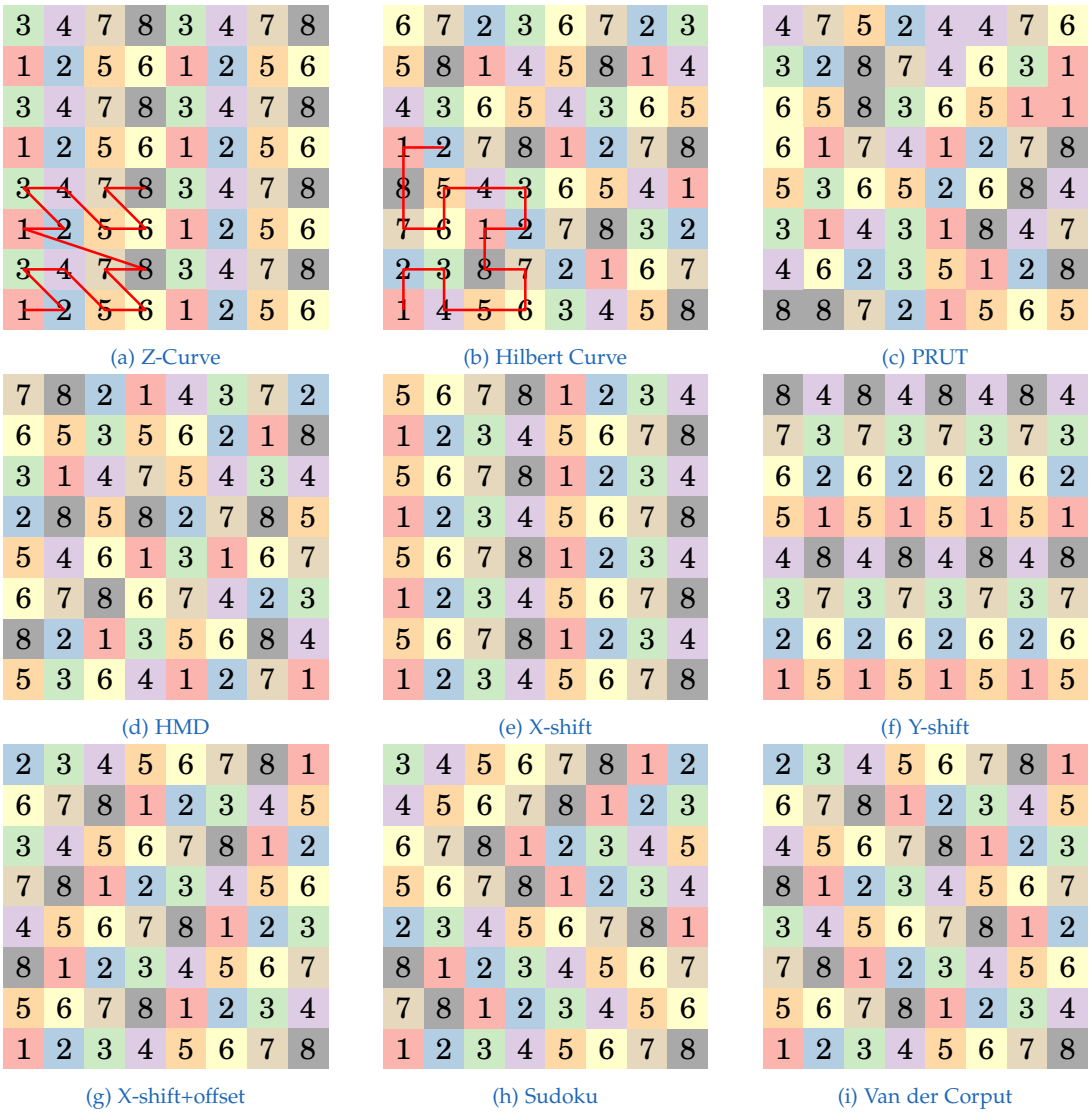
Based on our analysis, all applications show the highest variance in fragment load between lines oriented at fully horizontal and vertical orientations, thus confirming our initial assumption. With the exception of *Age of Mythology*, horizontal structures appear to have a much bigger impact than vertical ones. This is easily explained: *Age of Mythology* is the only application that enforces a fixed top-down view and has most objects of interest (*e.g.*, units, buildings and resources) vertically aligned. All remaining applications place the viewer at a first or third person perspective. From a viewpoint raised 1-2m above ground, far-reaching planar meshes (*e.g.*, terrain, water, floors, rooftops) tend to line up with the horizon, causing a significant concentration of geometry (and thus overdraw) at horizontal lines. Furthermore, most scenarios place complex objects on or close to a flat surface. Based on these insights, we formulate the goal for effective binning to avoid both vertical and horizontal rasterizer repetitions whenever possible.

7.4. Designing and Evaluating Patterns

In this section, we describe ten different patterns based on suggestions from previous work, adaptations of common space-filling techniques and our analysis of GPU hardware rasterization. Furthermore, we incorporate the insights gained in Section 7.3 in an effort to improve existing techniques and design a superior binning policy. We categorize, discuss and compare all patterns based on their basic approach. Finally, we pick the most promising pattern from each category to analyze trends and prospects for their expected effectiveness. All considered patterns (shown in Figure 7.6) are assessed using realistic clip-space geometry from our dataset.

Since an ideal binning policy would ensure completely uniform workload among all rasterizers, we rate patterns based on the variance of fragment load they produce. We process our input geometry data according to OpenGL conventions and record the amount and distribution of the resulting fragments, generated by the triangles processed in each rasterizer. In order to ensure that the observed trends are generally valid, we always assess variance at multiple bin sizes and rasterizer counts.

As a baseline for comparisons, we choose the *Diagonal* pattern (Figure 7.1d) used in several recent GPU models offered by Nvidia. This pattern can be generated as follows: Initially, all N rasterizers are lined up in ascending order according to their index. With each row, the index of the first rasterizer is



(a) Z-Curve (b) Hilbert Curve (c) PRUT

(d) HMD (e) X-shift (f) Y-shift

(g) X-shift+offset (h) Sudoku (i) Van der Corput

Figure 7.6.: Illustration of examined rasterizer patterns in a square bin tile of size 8. Bins of same color are assigned to the same rasterizer.

offset by one slot. Indices outside the possible range are wrapped around, creating a repetitive pattern. This policy leads to those bins that are assigned to a given rasterizer forming a diagonal line.

7.4.1. Space-filling Curves

Space-filling curves are a popular concept for efficiently querying and addressing multidimensional data. In computer graphics, popular applications include the creation of spatial data structures (Karras, 2012), as well as optimizing two-dimensional memory access when programming for the GPU (Nocentino and Rhodes, 2010). Here, we assess the performance of two space-filling curves that are well-established and routinely employed, namely the *Z-Curve* and the *Hilbert Curve*.

Z-Curve To produce a *Z-Curve* pattern that covers the entire viewport, we traverse all rasterizer bins and compute the 2D Morton code m_{xy} for each individual bin location (x, y), where $(0, 0) \rightarrow 0$ indicates the bin in the lower left corner. The index for the rasterizer to which we assign each bin is selected as m_{xy} mod N. Figure 7.6a shows the corresponding binning pattern, partially overlaid with the Z-Curve.

Hilbert Curve Similarly to *Z-Curve*, we traverse all bins in the viewport and use the 2D Hilbert distance function $dist_H(x, y)$ to compute the length of the curve at each bin location, with the bin in the lower left corner specifying the origin. The appropriate rasterizer is chosen from N available indices by calculating $dist_H(x, y)$ mod N. An exemplary layout following the Hilbert Curve in a square bin tile of size 8 is shown in Figure 7.6b.

Evaluation In Figure 7.7, we show the rasterizer load variance over our entire dataset for *Z-Curve* and *Hilbert Curve*. We plot the recorded values relative to our baseline *Diagonal* at different rasterizer counts. The thick stroke encompasses results for all tested bin sizes, ranging from 4×4 to 192×192 pixels, and thus indicates the influence of the bin size on the evaluation and the robustness of the pattern to changing this parameter. The thin opaque line marks the average over all bin sizes. Overall, both patterns appear to perform similarly well; however, *Z-Curve* behaves less consistently with varying bin sizes at low rasterizer counts, as indicated by the significant thickness of the red stroke. We thus consider *Hilbert Curve* the more suitable representative for expected performance from space-filling curves.

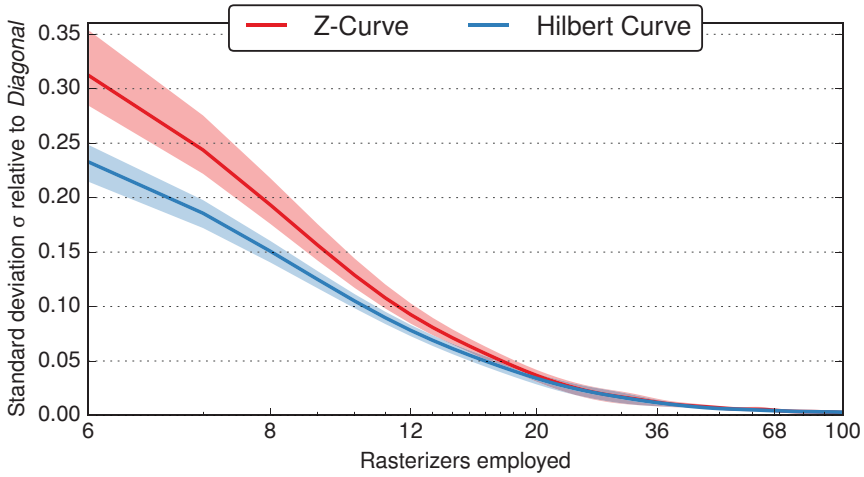


Figure 7.7.: Patterns using space-filling curves.

7.4.2. Randomized Patterns

In computer graphics, randomization is often used as a means to suppress noticeably repetitive artifacts or to create natural-looking shapes and patterns for visual scenes (*e.g.*, Monte Carlo methods). To assess randomization for our purposes, we examine two patterns whose layout is entirely defined by randomly generated values.

Pseudo-random Uniform Traversal This pattern is generated by traversing all bins over the image domain left-to-right, bottom-to-top and using the Mersenne Twister 19937 with a uniform distribution from 1 to N to choose a rasterizer for each bin at random. Hence, there are no guarantees ensuring minimum separation between bins assigned to the same rasterizer, or even equal occurrence of rasterizers throughout the image domain (Figure 7.6c).

Note that the space-filling curves and the pseudo-random uniform traversal are the only patterns we evaluate that may create a unique, non-repetitive arrangement over the entire viewport. All patterns discussed from here on out are defined by periodically repeating bin tiles.

Hierarchical Maximized Distance Inside a tile of $N \times N$ bins, we use randomized samples to assign rasterizers in a fashion similar to Poisson disk sampling. To fill all bins in the $N \times N$ tile, we cycle N times through the N

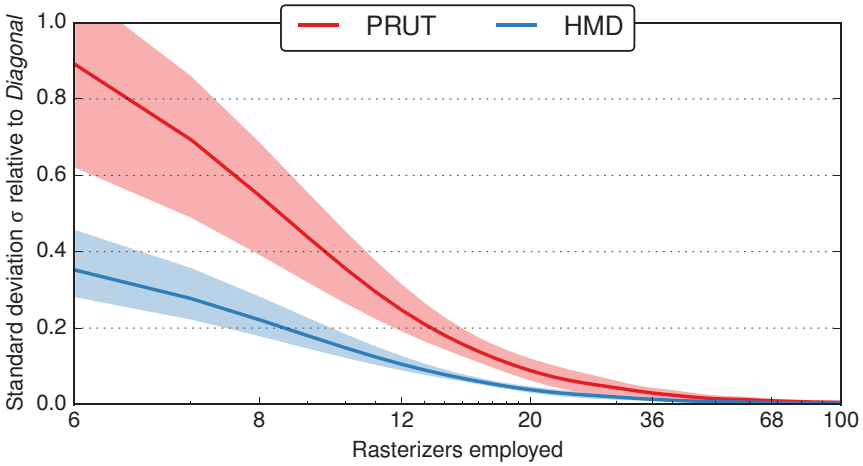


Figure 7.8.: Patterns based on randomization.

available rasterizer indices to ensure equal occurrence of all rasterizers in a tile. At each iteration, we use a simple dart-throwing technique with the rasterizer index as parameter R. We draw up to 50 vacant sample positions inside the $N \times N$ tile at random. In the absence of an ideal value for the Poisson disk radius to test against, we always select the bin that is the farthest from all other bins currently assigned to rasterizer R. The result of this process for an 8×8 tile is illustrated by Figure 7.6d.

Evaluation In Figure 7.8, we show the relative variance compared to our *Diagonal* baseline for *pseudo-random uniform traversal* (PRUT) and *hierarchical maximized distance* (HMD). *HMD* clearly outperforms *PRUT* on all accounts. We attribute the superior effectiveness to the fact that, in contrast to *PRUT*, *HMD* prioritizes large distances between bins assigned to the same rasterizer and thus encourages better space utilization.

7.4.3. Fixed Shift

Instead of shifting bin assignments by a single position in each row, as in the *Diagonal* pattern, we investigate the effect of wider fixed-distance shifts. Based on the trends we identified in the previous section, doing so is expected to significantly benefit from better space utilization and, consequently, more uniform distribution of localized geometry clusters to rasterizers.

X-Shift A horizontal shift for each row is computed by multiplying the row index r with a fixed value $\Delta = \frac{N}{k}$, where N is the number of rasterizers, and k gives the number of rows until the pattern repeats. The corresponding offset in row r can thus be computed as $\Delta_r = \left\lfloor \frac{r \cdot N}{k} \right\rfloor \bmod N$. Hence, *X-shift* forms rectangular, but not necessarily square periodically repeating $N \times k$ tiles (see Figure 7.6e). Notice that this implies a vertical rasterizer repetition every k rows. In order to maximize the distance between any two bins assigned to the same rasterizer, we choose k close to the square root of N as $\left\lfloor \sqrt{N} \right\rfloor$.

Y-Shift *Y-shift* is essentially a rotated version of *X-shift*. Shift parameters are chosen similarly. However, instead of a horizontal shift per row, a vertical shift is applied per column (see Figure 7.6f). Analogous to *X-shift*, *Y-shift* resets every k columns and has a tendency towards horizontal repetitions.

X-Shift+Offset While *Diagonal* has a small minimum distance between bins assigned to the same rasterizer, *X-shift* has a higher rate of rasterizer repetition on the vertical axis, making it a potentially weak candidate in rendering scenarios with prominent vertical structures (*e.g.*, *Age of Mythology*). *X-shift+offset* aims to combine the benefits of both approaches: based on the shift function for *X-shift*, row arrangements are offset by one position every k rows. The shift in row r can thus be computed as $\Delta_r = \left\lfloor \frac{r \cdot (N+1)}{k} \right\rfloor \bmod N$. Though this modification may be minor, it effectively ensures that vertical rasterizer repetitions do not occur before passing N rows (see Figure 7.6g). Thus, instead of periodic $N \times k$ tiles where $k \ll N$, *X-shift+offset* can fill a full $N \times N$ tile of bins before repeating itself.

Evaluation In Figure 7.9, we compare fixed shift patterns *X-shift*, *Y-shift* and *X-shift+offset* relative to *Diagonal*. When applied to our full dataset, *Y-shift* is clearly trailing behind the other alternatives. This comes as no surprise: According to the directional analysis of our dataset, horizontally aligned geometry is much more prominent; thus, the frequent horizontal rasterizer repetitions in *Y-shift* cause its performance to falter. An obvious exception to this observed trend is posed by *Age of Mythology*, for which the performance of *Y-shift* is generally on par and, in isolated cases, clearly better than *X-shift*. However, both patterns are outperformed by *X-shift+offset*, which we ascribe to the fact that *X-shift+offset* reduces both horizontal and vertical repetitions.

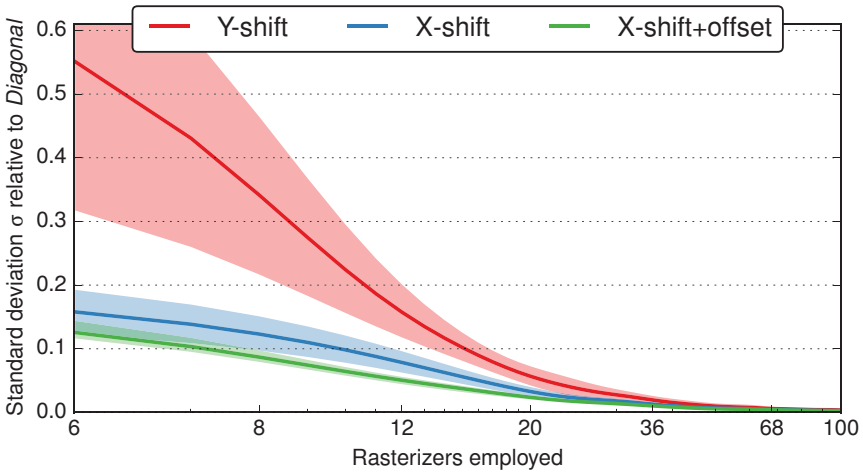


Figure 7.9.: Comparing fixed-shift patterns.

7.4.4. Variable Shift

In addition to patterns obtained by applying a fixed shift, we consider two instances of shift patterns with less predictable behavior.

Sudoku As part of their discussion on suitable binning patterns, M. W. Eldridge (2001) implicitly suggest that uniform workload distribution over periodically repeating $N \times N$ tiles can be facilitated by requiring that no two bins in the same row or column are assigned to the same rasterizer. Due to its conceptual similarity, we call this strategy *Sudoku*, after the popular Japanese puzzle. A random $N \times N$ tile that fulfills the *Sudoku* constraint can be quickly generated by drawing a random shifting value for each row in the interval $[0, N)$ and disallowing choosing the same shift value twice. Figure 7.6h shows one arrangement following the Sudoku policy in a square bin tile of size 8.

Van der Corput For this shift-based pattern, row shifts are computed based on a base 2 Van der Corput low-discrepancy sequence $\left(0, \frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{5}{8}, \frac{3}{8}, \ldots\right)$. The shift in row r is given by the r^{th} element in the sequence, multiplied by the next higher power of two for the number of rasterizers, $2^{\lceil \text{ld} N \rceil}$. Hence, the horizontal distance between two bins for the same rasterizer will be $> \frac{N}{2}$ for any N that corresponds to a power of 2. If this is not the case, all shifts $> N$ are skipped, generating a non-repeating pattern inside a bin tile of size

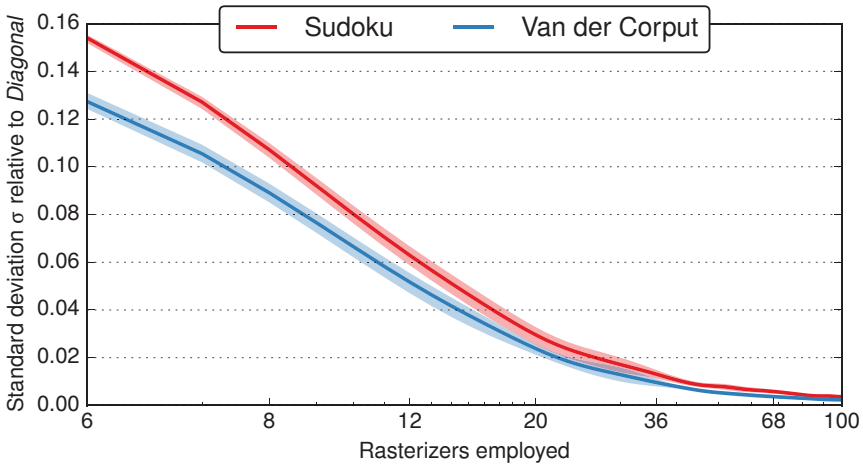


Figure 7.10.: Comparing variable-shift patterns.

N. Note that this pattern also implicitly fulfills the constraint that *Sudoku* is based on. However, there is no factor of randomization in its design. Figure 7.6i shows the definite arrangement of a square bin tile that is generated by this method for $N = 8$.

Evaluation Figure 7.10 compares the variance relative to *Diagonal* for *Sudoku* and *Van der Corput*. Both patterns show very similar development for varying rasterizer counts and bin sizes. We attribute this circumstance to their shared quality, namely the constraint of allowing rasterizers only once per row and column. Although both patterns appear to be very effective at distributing workload evenly, *Van der Corput* exhibits an evident advantage over *Sudoku*.

7.4.5. Comparison of All Categories

Finally, we compare the most promising patterns from all categories and assess their characteristics and overall behavior. Figure 7.11 shows the development of all approaches at three different bin sizes. In order to stress the possible benefits of using one pattern over another, we plot the coefficient of the variation $c_v = \frac{\sigma}{\mu}$. The choice of using c_v over σ is motivated by the fact that, in contrast to comparing quality of workload distribution, the standard deviation cannot adequately quantify the exact potential for improvement without knowing the average load per rasterizer μ. Trends for low rasterizer

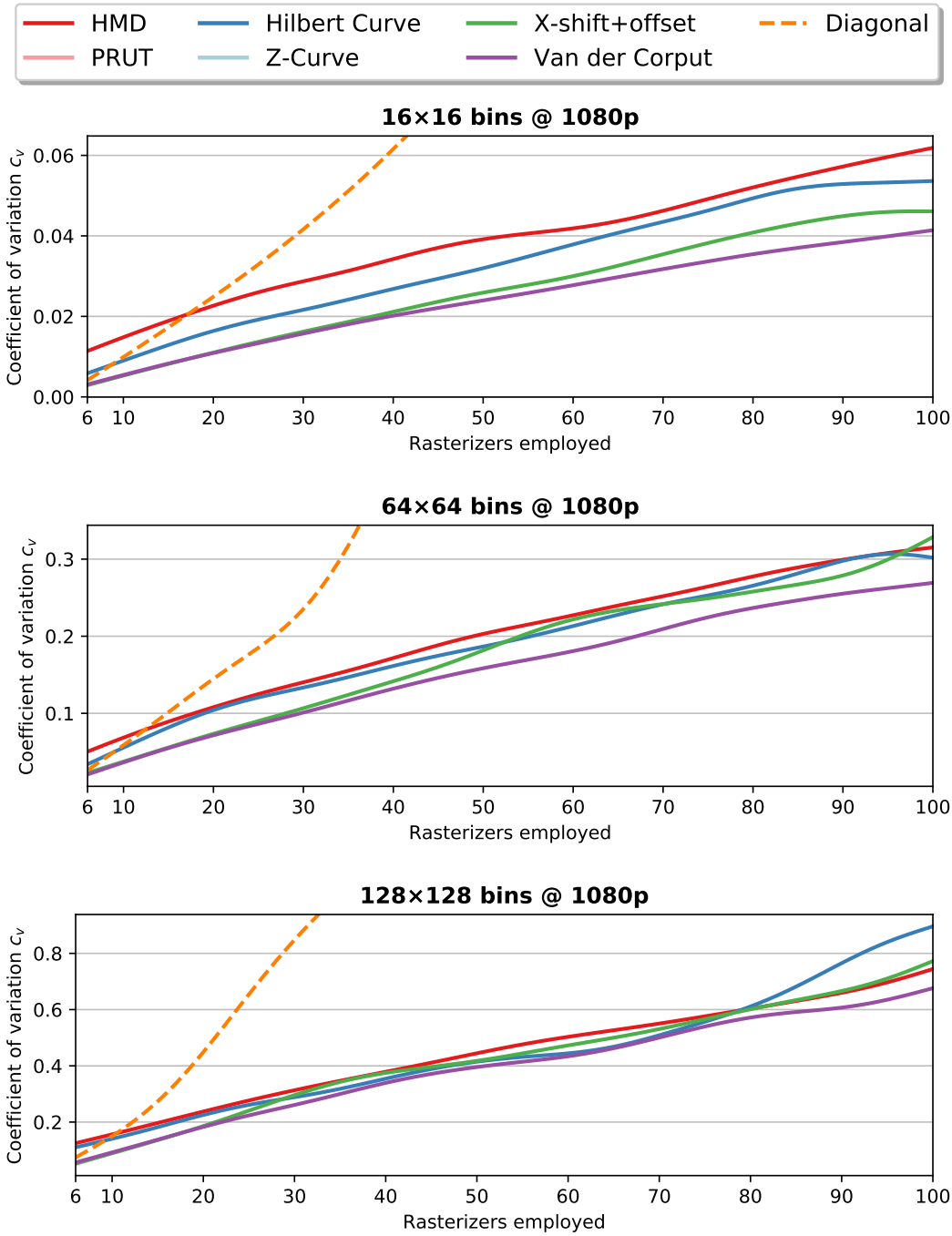


Figure 7.11.: Performance comparison on our dataset for the most promising patterns, alongside the *Diagonal* baseline. The coefficient of variation c_v estimates the imbalance in rasterizer workload for each pattern at different configurations. While *X-shift+offset* performs as well as *Van der Corput* for low rasterizer counts, they usually start to diverge at ∼20 rasterizers.

counts are identical at all evaluated bin resolutions: *Diagonal* quickly falls behind all other techniques due to its poor handling of clusters and space utilization. All other patterns exhibit a much slower growth of c_v, with *X-shift+offset* and *Van der Corput* tied for best performance. However, for higher bin resolutions and rasterizers counts, *X-shift+offset* gradually falls behind. For bigger bin sizes, the differences between the techniques (with the exception of *Diagonal*) become less pronounced. Note however the ranges of the plotted coefficients for the respective bin sizes: The recorded values of c_v at 128×128 differ from those at 16×16 by more than one order of magnitude. Thus, differences between the individual techniques have a much higher impact at bigger bin sizes in terms of performance. The most promising pattern out of those evaluated is *Van der Corput*, with its coefficient of variation consistently below or on par with its contenders.

Considering our previously stated guidelines for pattern design, the high performance of *Van der Corput* is not surprising. In each 2D bin tile ($N \times N$), each rasterizer is referenced with the same frequency, leading to a good space utilization. Within each row, the distance between bins assigned to the same rasterizer is maximal. The same is true for each individual column. Thus, horizontally and vertically dense regions will be assigned to the same rasterizer only when there is no way to avoid that. Moreover, the pattern generation rule ensures that the 2D distance between rasterizers is always high, avoiding local clusters. The direction in which rasterizer assignments repeat within a 2D region is loosely oriented along a 45° angle. All these considerations also apply to *X-shift+offset*, which also shows a very competitive performance. However, *Van der Corput* is locally less structured than *X-shift+offset*, which probably is the reason for giving it an additional edge over the other approaches, especially when the rasterizer count increases.

7.4.6. Influence of Partitioning

So far, we have evaluated patterns on typical GPU workloads for complete frames. However, when rendering large scenes, the GPU cannot process all primitives at once and only a limited number of primitives are concurrently in-flight. In order to quantify the influence of this workload partitioning in the context of binning patterns, we split the input triangle stream of each scene into M batches of equal size, while maintaining the order of primitives as they were submitted to the GPU. We then simulate the binning of each individual batch separately, evaluate the resulting temporally local variance,

and compute an average c_v from all M individual batches. By performing this process with several different values for parameter M, we can identify trends that are likely to influence pattern performance on actual hardware.

Figure 7.12a shows the influence of the batch size on c_v when using 6 rasterizers and a bin size of 16×16. This is equivalent to the configuration used in the Nvidia GeForce Titan Xp and thus representative for current graphics hardware. For very small batch sizes, the relative variance is high, as only few triangles are rendered and few bins actually receive workloads. However, as the batch size increases to $10 - 25\%$ of the full scene load, c_v already converges to the figures measured for the full scene. This points towards patterns already performing as expected when only a small portion of the scene can be processed in parallel. As the amount of rendered geometric detail is steadily increasing, this result also points towards the necessity to be able to process at least 10% of a scene at once to achieve close-to-ideal load balancing. In Figure 7.12b, we consider a setup with a higher number of simulated rasterizers. This would correspond to, *e.g.*, a software rendering approach on the GPU, since the GPCs are inaccessible. The largest independent structure that can be explicitly controlled and programmed on the GPU are instead its shared multi-processors. The choice to set the number of rasterizers to 20 reflects an approximation of available processing power on contemporary GPU models. More specifically, this setup corresponds to a software renderer supplying one rasterizer per SM on a GTX 1080Ti. We can see that the variance remains higher for a longer period of time, starting to converge at roughly 20% of the scene being included in one batch.

We further consider the expected benefit of choosing one pattern over another for these smaller workloads. To visualize their relative behavior in detail, we plot the development of c_v as the degree of partitioning increases relative to *Van der Corput* as a reference in Figure 7.12c. The relative difference in performance between patterns steadily decreases as the batch size becomes smaller. However, when using batches with a size of $1/100$ of the full scene, the difference between the best and worst pattern still makes for a factor of $2\times$. Like in our previous experiments, both *X-shift+offset* and *Van der Corput* remain the most efficient methods, even though the initial advantage of *X-shift+offset* degrades with batches getting smaller. Surprisingly, we found that *HMD* approaches more advanced patterns with increasing degree of partitioning, and even reaches sophisticated space-filling curves. This can be explained by the fact that smaller batches affect a limited portion of the screen and *HMD* explicitly considers separation of rasterizers in its layout, which is the most dominant factor when rendering few primitives.

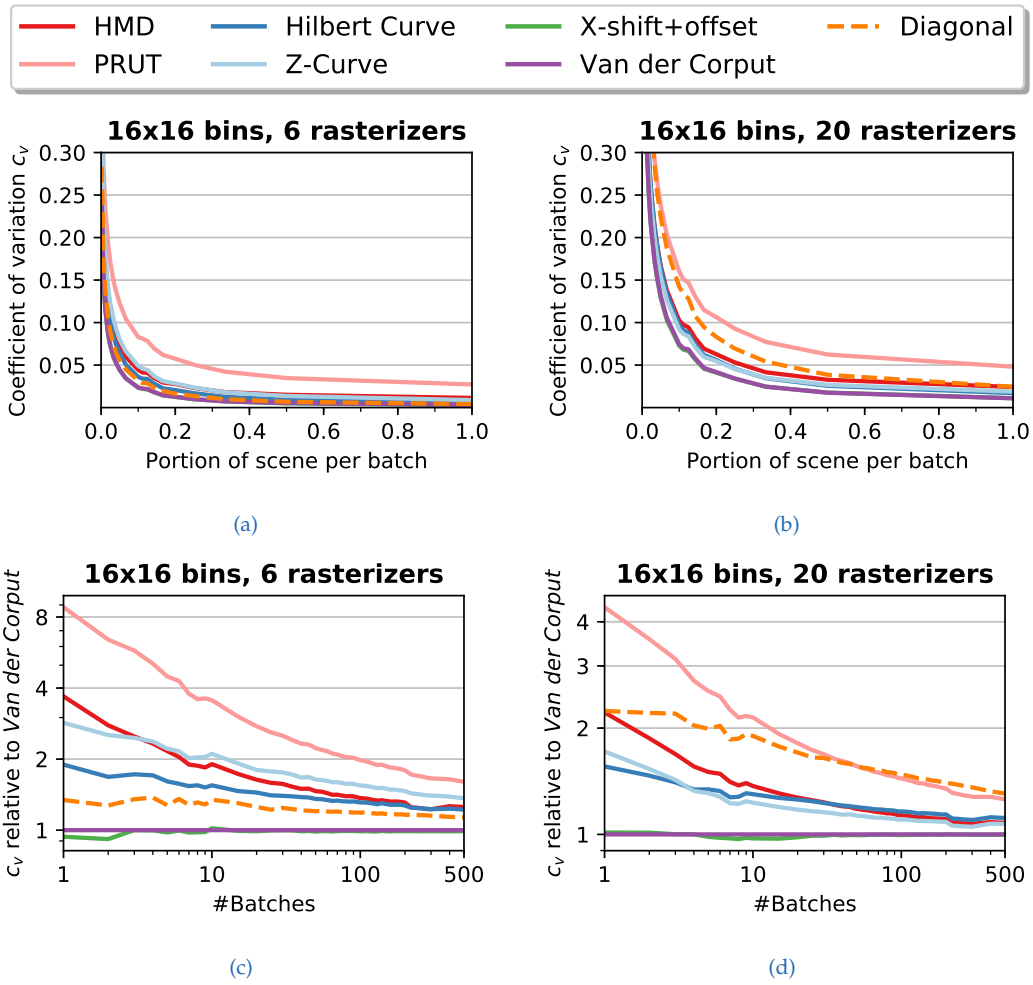


Figure 7.12.: Developments in pattern performance with regard to scene partitioning into batches. (a) For configurations corresponding to current hardware rendering, partitioning strongly affects c_v and thus the expected quality of load balancing. Measurements converge toward previous results at a batch size of about 1/10 of the scene. (b) For the parallelism that can be expected when using one bin per SM in a software renderer, the convergence rate is considerably slower. (c) Relative to *Van der Corput*, *HMD* converges toward similar performance as filling curves. As with increasing the number of rasterizers, partitioning causes *Diagonal* to eventually deteriorate. Configuration (d) shows *Diagonal* being overtaken by the trivial *PRUT*, and *Z-Curve* outperforming *Hilbert Curve*.

As can be seen in Figure 7.12d, increasing the number of rasterizers to 20 results in a stronger influence of partitioning on *Diagonal*. While *Diagonal* performs equally well as *HMD* when processing the entire scene at once, it fails to show relative improvement and is eventually even overtaken by the straightforward *PRUT* pattern. Thus, as the size of batches decreases, *Diagonal* eventually falls behind all other alternatives. Furthermore, we find this setup to be one of several instances where *Z-Curve* eventually beats *Hilbert Curve*. This is not a general rule, but still a common occurrence, which solidifies our initial impression that space-filling curves should be considered on a case-by-case basis, since there is no single best choice for all configurations.

Finally, we found that partitioning favorably affects *X-shift+offset* when using bigger bin sizes with high rasterizer counts. In those large-bins/many-rasterizers configurations, as shown above, the pattern usually starts to trail behind *Van der Corput* when processing entire scenes. However, increasing the number of batches a scene is partitioned into causes *X-shift+offset* to quickly approach the reference. We consider this circumstance an argument for the general usability of *X-shift+offset*, as its divergence from *Van der Corput* appears to be mitigated by the (likely) partitioning of large scene input data.

7.4.7. Observations and Remarks

The coefficient of variation (see Figure 7.11) lets us speculate on how the employed patterns may translate into performance on real hardware. For a bin size of 16×16 @ 1080p, where a bin covers $1/120$ of the screen horizontally, up to 18 rasterizers achieve a $c_v < 1\%$ using the best pattern. This setup corresponds to the bin size employed on current Nvidia GPU designs, which often rely on a diagonal pattern. Translated into runtime performance, rasterizers must handle a load imbalance of less than 1% on average. Considering that there might be other system-wide load balancing strategies happening concurrently, for instance, with vertex processing on the GPU, one would probably not expect any noticeable performance hit. In comparison, if a diagonal pattern is used, the 1% threshold is already exceeded with 10 rasterizers. We hypothesize that the challenges of finding a static pattern that ensures equally uniform load balancing across many processors is one reason for using relatively few logical rasterizers (GPCs) in current GPU designs, even as the SM count is increased. A consequence of this design choice is that more advanced dynamic load balancing strategies between a rasterizer and its associated multiprocessors are needed.

Naturally, the demands on the rasterizer itself also increase as the resolution increases. Since the graphics pipeline must enforce primitive order during rasterization (Purcell, 2010), the rasterizer is not completely free in its strategies for parallelization. Thus, it is questionable whether rasterization performance can be scaled along with fragment processing performance in future hardware designs without increasing the number of rasterizers.

With larger bin sizes, the influence of the pattern becomes more severe. In these cases, our best contender, the *Van der Corput* pattern, deviates from an ideal distribution by up to 10% for 30 rasterizers with a bin size of 1/30 of the viewport width, and 25% with a bin size of 1/15. Such larger bin sizes relative to the viewport width might be found on mobile devices, which render in lower resolution, or in software-based rendering, which tries to avoid communication overhead between compute cores.

Analyzing the influence of scene partitioning on workload variance, our experiments indicate that about $10 - 20\%$ of the scene should be processed in parallel. In this case, we see an equally good work distribution as if the entire scene was processed as one. When processing only small geometry batches, the relative performance difference between the patterns hardly changes, indicating that our pattern design criteria are largely independent of the amount of data being rendered. This is not surprising, as most considerations guide local bin assignment rather than a global strategy.

It is tempting to draw the conclusion that a small bin size is essential for good performance. However, our analysis only considers the load balancing characteristics and ignores the cost of data transfer and duplication. A smaller bin size will quickly lead to increased communication overhead, as the same triangle now overlaps more bins and is assigned to more rasterizers. Depending on the implementation, this overhead may effectively outweigh the benefits of using smaller bins.

7.5. Binning Patterns for Software Rasterization

In order to test our pattern designs in a complete system and to verify theoretical results for their impact on rendering performance, we use CURE, our software rendering pipeline running in CUDA, which supports arbitrary patterns for rasterization (Kenzel, Kerbl, Schmalstieg et al., 2018). Note that CURE follows a full streaming model, employs a sort-middle design, and can

dynamically switch between tasks in the geometry stage and rasterization/-fragment stages. Furthermore, it does not process entire scenes at once but gradually streams its pipeline data, overall mimicking hardware rendering. Thus, this experimental setup gives an accurate estimate for the performance impact of tested patterns. We have extended the original input geometry with auxiliary data listing the appropriate rasterizer indices for every triangle. We precompute these data for all test scenes, bin sizes and patterns and load them during rendering to perform load balancing according to each pattern.

We process our test dataset at two different bin sizes of 16×16 and 64×64 pixels and assess pattern performance for rendering at 1080p resolution. As we do not have access to the hardware rasterizers (GPCs) directly, our implementation employs an equal number of rasterizers on each SM in software. Thus, the rasterizer count must be either smaller than or a multiple of the SM count, and we run our tests using 6, 20 and 60 logical rasterizers on a GTX 1080Ti. We simulate elaborate fragment shading by performing 2500 fused multiply-add instructions before submitting the fragment color.

Table 7.2 shows the average achieved frame rates as frames per second (FPS) for running the full test set with each technique at different bin sizes and rasterizer counts. The numbers in brackets further state the harmonic mean of the average relative speed-up in each scene over the baseline set by *Diagonal*. The behavior of the patterns at different settings closely matches our predictions: with higher numbers of rasterizers and bigger bin size, the impact of choosing a sophisticated pattern increases, outclassing *Diagonal* by a factor of almost $1.9\times$ using 60 rasterizers at 1080p. The relative performance gain between different patterns is also amplified with the rise of either parameter. Overall, we see that both *X-shift+offset* and *Van der Corput* perform the strongest, with the former dominating at low and the latter at high rasterizer counts.

A detailed breakdown of the times spent on the individual pipeline stages by each SM for the game scene in Figure 7.2b is shown in Figure 7.13. For the diagonal pattern, it can be observed that the dynamic load balancing between geometry processing and rasterization/fragment processing clearly moves the geometry load to those SM instances that receive the least rasterization load (SM 3-10). However, this trivial load balancing is not sufficient to counteract the imbalance introduced by *Diagonal*, and a majority of the SM instances (1-17) idle for about 15% of the total runtime.

Table 7.2.: Results for tested patterns on our dataset at multiple configurations. For each pattern, we show average achieved frame rate, as well as relative speed-up over *Diagonal*. The technique with best performance is marked bold for every setup.

(a) 16 × 16 bin size

#Rasterizers	Hilbert	HMD	X-shift+offset	Van der Corput
6	3.3 (1.00)	3.3 (1.00)	**3.3 (1.00)**	3.3 (1.00)
20	10.9 (1.00)	10.9 (1.00)	**11 (1.01)**	11 (1.01)
60	16.2 (1.09)	16.1 (1.08)	16.4 (1.10)	**16.4 (1.10)**

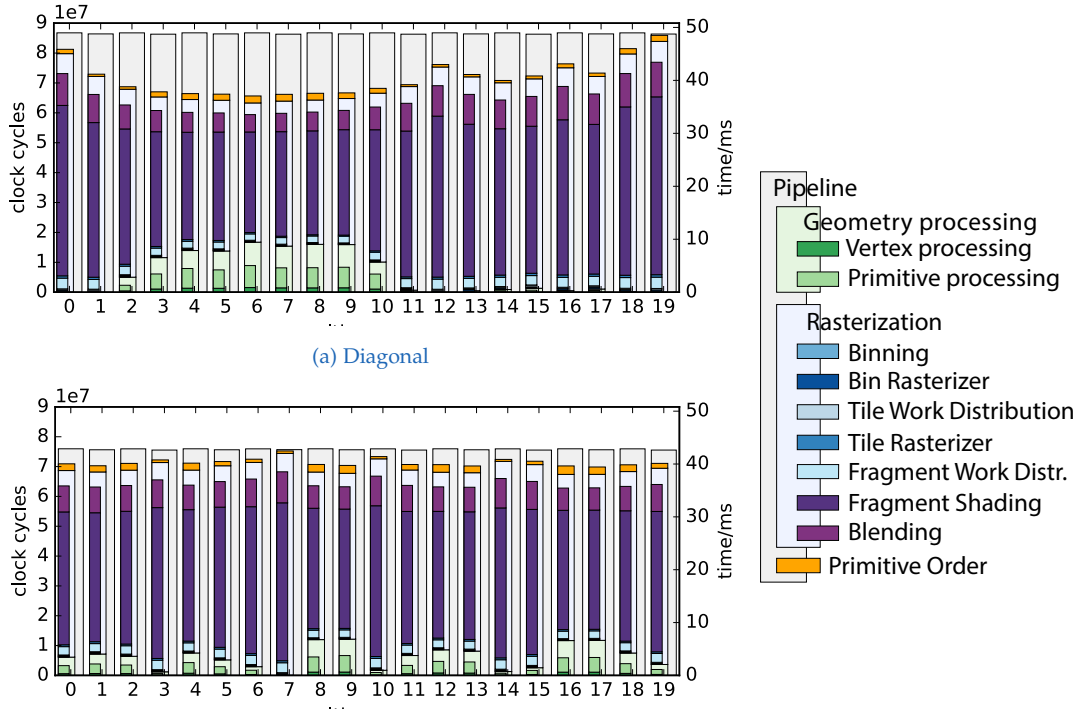
(b) 64 × 64 bin size

#Rasterizers	Hilbert	HMD	X-shift+offset	Van der Corput
6	3.3 (0.99)	3.3 (0.99)	**3.4 (1.01)**	3.4 (1.01)
20	9.5 (1.02)	9.6 (1.04)	**10.3 (1.12)**	10.3 (1.11)
60	10.6 (1.72)	10.3 (1.70)	11.0 (1.78)	**11.6 (1.89)**

Using *Van der Corput*, the rasterization load is distributed more uniformly, and the remaining imbalance is counteracted by the dynamic load balancing with geometry processing. All SMs finish within 4% of the runtime, significantly boosting overall performance. Note that, in actual game content, the relative load on geometry processing and fragment processing is often even more skewed than in our test example, as fragment shaders tend to become more and more complex.

7.6. Discussion

We have identified and analyzed possible influential factors on load balancing and overall performance to be considered when designing a static binning pattern for sort-middle rasterization. In an effort to optimize load balancing behavior, we have presented several different examples of patterns with distinct characteristics and assessed them both analytically and practically. Runtime measurements for each applied pattern in various configurations were obtained using CURE, a state-of-the-art software rendering pipeline for the GPU. Based on our predictions and their confirmation from the measured runtime results, we have successfully identified a set of patterns that scale well with the number of rasterizers and can exhibit significantly improved performance over naïve approaches.



(a) Diagonal

(b) Van der Corput

Figure 7.13.: Detailed performance breakdown of the rendering pipeline runtime using Diagonal and Van der Corput patterns with 20 rasterizers at 1080p to draw the scene from *Rise of the Tomb Raider* shown in Figure 7.2b in CURE. While dynamic load-balancing between geometry processing (green) and Rasterization (purple) can counteract different fragment loads to some extent, the imbalance created by Diagonal is too severe, and the overall occupancy clearly suffers, with many processors being idle for more than 15% of the total runtime.

Specifically, we have identified two deterministic patterns that exhibit close-to-ideal behavior and are easy to apply. Although performance gains may be negligible for small bin sizes and low rasterizer counts, they become more pronounced when either of these values grows. The desire for high resolutions and the communication penalty of small bin sizes is likely to lead to the utilization of more rasterizers, in turn creating a strong need for a good binning pattern in new hardware designs. However, even if future developments were to solve this problem in a different way, current software rendering applications like CURE that are barred from access to low-level hardware features (*e.g.*, GPCs) can already exploit our findings to benefit from improved load balancing today, as we have shown.

8. Conclusion

Contents

8.1. Summary

In this thesis, we strove to put dynamic and static load balancing approaches on the GPU to the test. We have considered the GPU in its two most relevant contexts: as a ubiquitous, increasingly programmable and massively parallel co-processor to the sequentially oriented CPU; and as a highly specialized hardware pipeline for sort-middle rendering. For the first, we were driven by recent achievements regarding dynamic load balancing to devise a novel, general solution for high-performance custom scheduling and prioritization.

We have dissected the individual components that previous scheduling systems were based on and identified concurrent queuing algorithms as pivotal performance factors. To cater to the particular needs and peculiarities of the GPU architecture, we have designed a linearizable queuing algorithm, the broker queue, that fulfills the conceivable requirements for optimal performance in this domain. Our evaluation has shown that the resulting queue is several orders of magnitudes faster than other linearizable alternatives, and even outperforms non-linearizable, potentially blocking algorithms.

For both synthetic and realistic workloads, the broker queue and its variants seem to be the most suitable choice to date for running GPU applications that aim to surpass the rigid hardware scheduling model and supply dynamic load balancing instead. Its support for fair ordering, as well as multi-queue structures, further renders them prime candidates to be used in higher-level scheduling systems.

With the availability of these exceedingly fast queuing methods under our belt, we revisited the idea of sorting work packages for larger GPU compute jobs, according to their defined importance for progress in the big picture. Unwilling to sacrifice either performance or fine-granular categorization, we have proposed a middle-of-the-road solution: hierarchical bucket queuing. We can deploy arbitrarily complex scheduling strategies with this approach, limited in scope only by our skill and on-device memory. The granularity for prioritization in a hierarchy can be customized to address particular needs for load balancing. This has been demonstrated in-depth on fundamental scheduling tasks, proving that quota-based and earliest-deadline-first problems pose no problem for our scheduling framework. In terms of versatility and efficacy, it easily surpasses relevant previous approaches. The exhibited performance is sufficient to apply hierarchical bucket queuing even to the time-critical procedures found in software rendering pipelines and can introduce adaptive behavior with the help of efficient task prioritization.

While prioritized software rendering remains an enticing project for optimal exploitation of the GPU's raw computing power, the GPU is also subject to several intricate load balancing strategies in its original role as a hardware rasterizer. We have taken a closer look at two specific stages of the conventional rendering pipeline, namely vertex transformation and rasterization. In both cases, we have found that a rigid distribution of workload is usually employed on top of trivial dynamic load balancing, in order to leverage vertex data reuse and sort-middle primitive-to-rasterizer assignment.

The consideration of how workload can be effectively distributed in a massively parallel system has brought us to the logical—and ultimately verifiable—conclusion, that the idea of a centralized post-transform vertex cache is no longer accurate. Instead, we have identified static procedures used to split input streams into batches with predictable vertex reuse to satisfy demands for load balancing as well as avoiding redundant shader invocations. The subsequent assignment in the graphics pipeline of output geometry data to image-space bins is required in the sort-middle architecture to allow rasterizers exclusive access to render targets. We have identified patterns that are

commonly used by GPUs to achieve this feat and provided our own pattern designs and evaluations thereof. Incorporating the most effective pattern into CURE, our parallel software rendering pipeline has caused a noticeable reduction in runtime, especially for fragment-heavy routines.

8.2. Observations and Insights

With regard to GPU queuing, we have seen that the benefits of elaborate algorithms, *i.e.*, achieving continuous, system-wide progress, can easily be outweighed by the implied overhead. Furthermore, the high resource consumption of complex procedures can lead to reduced occupancy, since the limitations on registers and shared memory are quickly exhausted by elaborate functions and structures. We have seen this specifically in the cases of the LCRQ and the WFQ during our evaluation. This implies that GPU frameworks must put particular focus on keeping functions light-weight. A seemingly naïve approach—such as the reliance of the broker queue on contended atomic operations—can, in fact, be much more effective than building complex methods that rely on optimistic concurrency control.

In much the same spirit ('less is more'), we found that for hierarchical bucket queuing, a limited number of discrete priority ranges can already elicit adaptive behavior in scheduling frameworks. With approximately 16 bucket queues being used, we have already observed prioritization behavior with an accuracy close to what can be ideally expected in such a parallel system. Further increase in the granularity achieves only minor improvements and causes additional overhead during hierarchy traversal. Furthermore, the idea (and system support) to interrupt megakernels and immediately reuse queues without initializing is a powerful one, since it enables running time-constrained procedures, *e.g.*, real-time rendering applications. Even though it now seems feasible that custom dynamic load balancing for software rendering pipelines on the GPU may well be within our grasp, we must also not ignore the particular use cases where static load balancing remains essential.

Static load balancing must, for instance, be considered when revisiting allegedly established concepts of GPU hardware rendering. Based on the prevailing assumption that the post-transform cache is an indispensable part of the rendering pipeline, mesh optimization algorithms have aimed to produce reordered sequences of vertex references that cater to this particular hardware module. We have investigated the measurable distribution of concurrently

processed vertices and thereby come to the conclusion that hardware behavior does not match the vertex cache theory. While there appears to be limited retention of previously computed results, vertex reuse is neither global nor dynamic. On recent GPU architectures, our batch-based reuse models are more accurate at predicting average shading rates than the cache-based alternatives. Knowing the exact batching function of a device enables us to present our own variation of reordering-based mesh optimization for vertex reuse. On Nvidia models, we claim to have in essence deciphered all relevant parts of the batching function: here, our batch-based optimization technique was shown to be the most effective method available.

Also on Nvidia GPUs, we have identified the go-to solution for sort-middle primitive-to-rasterizer assignment to be a simple diagonal pattern, or a slight variation thereof. While our evaluation shows that the diagonal pattern is not the most scalable for balancing rasterizer load by a long shot, the fact that GPUs continue to employ only a small number of graphics processing clusters for rasterization relativizes this verdict: at roughly 5–7 rasterizers being used in recent models, the impact of choosing a more suitable pattern is, according to our simulations, marginal at best. Once again, the recurring theme for the findings in this thesis appears to be: 'Less is more'. However, these insights may become relevant as the demand for ever higher parallelism increases in future hardware generations. Furthermore, control of the dedicated GPU rasterization units is exclusive to the hardware pipeline and is not available when using compute APIs and software solutions. For those, our experiments have shown that the use of an appropriate static rasterizer pattern can significantly improve performance in a state-of-the-art sort-middle software renderer and produce equalized load.

8.3. Future Work

While the contributions presented in this thesis provide several revelations and insights on suitable mechanisms for load balancing on the GPU, there are still open questions and unsolved problems that inhibit their applicability. For one, the adaptive rendering solutions presented in the context of hierarchical bucket queuing are comparatively simple in their design. This was not necessarily a choice: more elaborate adaptive sampling and subdivision schemes usually imply non-local dependencies to, *e.g.*, compare local error estimates with bordering image regions. Such a concept could, in our current system,

only be implemented through the use of costly global synchronization and communication routines, which would severely impact performance. Thus, a useful future extension would be additional support for queue communication, enabling algorithms to check and possibly reassign priorities across several layers of a hierarchy.

The mesh optimization algorithm that we presented to fit batch-based workload distribution performs exceptionally well on all tested Nvidia models. However, on other architectures, results are more modest, as for AMD and Intel, the identification of the respective batching function is incomplete. Unfortunately, this is in part due to the short-term unavailability of adequate hardware to us and curt research project time frame. Needless to say, with additional time and labor, it should be possible to create a well-rounded solution that produces ideal vertex streams for all contemporary devices. The creation of an accessible, easy-to-use optimization suite is an important item on our agenda and will be pursued to completion in the near future.

It should also be noted that, although we have identified new and suitable rasterizer patterns, their elaborate design makes them less trivial to compute than the simpler alternatives. For an operation that occurs as frequently in a rendering pipeline as sorting a primitive into a screen-space bin, the architecture should neither have to rely on complex computations nor on a lookup of a precomputed pattern layout from memory. So far, we have not found a trivial solution to compute covered bin indices for primitives with our suggested patterns while ensuring a low code and resource footprint. However, we are making steady progress and hope to provide an optimal implementation of this pattern in our GPU software renderer, CURE, soon.

Finally, we aim to conduct further research into applications of load balancing, as we continue to refine and extend the features in various GPU-related projects. The best place to apply any new knowledge will be the ongoing CURE project, where we aim to keep pushing the boundaries of high-performance software rasterization. We hope to present new, relevant contributions soon in the context of CURE and any of its eventual descendants.

Bibliography

Aila, Timo and Samuli Laine (2009). 'Understanding the efficiency of ray traversal on GPUs'. In: *Proc. High Performance Graphics*. HPG '09. New Orleans, Louisiana: ACM, pp. 145–149. ISBN: 978-1-60558-603-8. DOI: 10.1145/1572769.1572792. URL: http://doi.acm.org/10.1145/1572769.1572792 (cit. on pp. 9, 78).

AMD (2012). *White Paper: AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE*. https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf. Retrieved June 16, 2017 (cit. on p. 108).

Arora, Nimar S., Robert D. Blumofe and C. Greg Plaxton (1998). 'Thread scheduling for multiprogrammed multiprocessors'. In: *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*. SPAA '98. Puerto Vallarta, Mexico: ACM, pp. 119–129. ISBN: 0-89791-989-0. DOI: 10.1145/277651.277678. URL: http://doi.acm.org/10.1145/277651.277678 (cit. on pp. 10, 42).

Barczak, Joshua (2016). *Vertex Cache Measurement*. Retrieved: June 4th, 2018. URL: http://www.joshbarczak.com/blog/?p=1231 (cit. on pp. 85, 86, 91).

Basaran, C. and Kyoung-Don Kang (July 2012). 'Supporting Preemptive Task Executions and Memory Copies in GPGPUs'. In: *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pp. 287–296. DOI: 10.1109/ECRTS.2012.15 (cit. on p. 8).

Bell, Nathan and Michael Garland (2009). 'Implementing sparse matrix-vector multiplication on throughput-oriented processors'. In: *Proc. High Performance Computing Networking, Storage and Analysis*. SC '09. Portland, Oregon: ACM, 18:1–18:11. ISBN: 978-1-60558-744-8. DOI: 10.1145/1654059.1654078. URL: http://doi.acm.org/10.1145/1654059.1654078 (cit. on p. 9).

Blelloch, Guy E., Perry Cheng, Phillip B. Gibbons and P. B. Gibbons (2003). 'Theory of Computing Systems Scalable Room Synchronizations'. In: (cit. on pp. 11, 32).

Breitbart, Jens (2011). 'Static GPU threads and an improved scan algorithm'. In: *Proc. Conference on Parallel Processing*. Euro-Par 2010. Ischia, Italy: Springer-Verlag, pp. 373–380. ISBN: 978-3-642-21877-4. URL: http://dl.acm.org/citation.cfm?id=2031978.2032029 (cit. on p. 9).

Cater, Kirsten, Alan Chalmers and Patrick Ledda (2002). 'Selective Quality Rendering by Exploiting Human Inattentional Blindness: Looking but Not Seeing'. In: *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*. VRST '02. Hong Kong, China: ACM, pp. 17–24. ISBN: 1-58113-530-0. DOI: 10.1145/585740.585744. URL: http://doi.acm.org/10.1145/585740.585744 (cit. on p. 75).

Catmull, E. and J. Clark (1998). 'Seminal Graphics'. In: New York, NY, USA: ACM. Chap. Recursively Generated B-spline Surfaces on Arbitrary Topological Meshes, pp. 183–188. ISBN: 1-58113-052-X. DOI: 10.1145/280811.280992. URL: http://doi.acm.org/10.1145/280811.280992 (cit. on p. 13).

Cederman, Daniel and Philippas Tsigas (2008). 'On dynamic load balancing on graphics processors'. In: *Proce. Symposium on Graphics Hardware*. GH '08. Sarajevo, Bosnia and Herzegovina: Eurographics Association, pp. 57–64. ISBN: 978-3-905674-09-5. URL: http://dl.acm.org/citation.cfm?id=1413957.1413967 (cit. on pp. 9, 30).

Chatterjee, Sanjay, Max Grossman, Alina Sbirlea and Vivek Sarkar (2011). 'Dynamic Task Parallelism with a GPU Work-Stealing Runtime System'. In: *Proc. Languages and Compilers for Parallel Computing*. LCPC '11 (cit. on pp. 9, 30).

Chen, Jiawen, Michael I. Gordon, William Thies, Matthias Zwicker, Kari Pulli and Frédo Durand (2005). 'A Reconfigurable Architecture for Load-balanced Rendering'. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '05. Los Angeles, California: ACM, pp. 71–80. ISBN: 1-59593-086-8. DOI: 10.1145/1071866.1071878. URL: http://doi.acm.org/10.1145/1071866.1071878 (cit. on p. 17).

Chen, Long, O. Villa, S. Krishnamoorthy and G.R. Gao (2010). 'Dynamic load balancing on single- and multi-GPU systems'. In: *Parallel Distributed Processing (IPDPS)*, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470413 (cit. on pp. 9, 58).

Chen, Milton, Gordon Stall, Homan Igehy, Kekoa Proudfoot and Pat Hanrahan (1998). 'Simple Models of the Impact of Overlap in Bucket Rendering'. In: *SIGGRAPH/Eurographics Workshop on Graphics Hardware*. Ed. by S. N. Spencer. The Eurographics Association. DOI: 10.2312/EGGH/EGGH98/105-112 (cit. on p. 17).

Chhugani, Jatin and Subodh Kumar (2007). 'Geometry Engine Optimization: Cache Friendly Compressed Representation of Geometry'. In: *Pro-*

ceedings of the 2007 Symposium on Interactive 3D Graphics and Games. I3D '07. Seattle, Washington: ACM, pp. 9–16. ISBN: 978-1-59593-628-8. DOI: 10.1145/1230100.1230102. URL: http://doi.acm.org/10.1145/1230100.1230102 (cit. on p. 16).

Chow, Mike M. (1997). 'Optimized Geometry Compression for Real-time Rendering'. In: *Proceedings of the 8th Conference on Visualization '97*. VIS '97. Phoenix, Arizona, USA: IEEE Computer Society Press, 347–ff. ISBN: 1-58113-011-2. URL: http://dl.acm.org/citation.cfm?id=266989.267103 (cit. on p. 15).

Clarberg, Petrik, Robert Toth and Jacob Munkberg (July 2013). 'A Sort-based Deferred Shading Architecture for Decoupled Sampling'. In: *ACM Trans. Graph.* 32.4, 141:1–141:10. ISSN: 0730-0301. DOI: 10.1145/2461912.2462022. URL: http://doi.acm.org/10.1145/2461912.2462022 (cit. on p. 16).

Colvin, R. and L. Groves (June 2005). 'Formal verification of an array-based nonblocking queue'. In: *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pp. 507–516. DOI: 10.1109/ICECCS.2005.49 (cit. on p. 32).

Cook, Robert L., Loren Carpenter and Edwin Catmull (Aug. 1987). 'The Reyes image rendering architecture'. In: *SIGGRAPH Comput. Graph.* 21.4, pp. 95–102. ISSN: 0097-8930. DOI: 10.1145/37402.37414. URL: http://doi.acm.org/10.1145/37402.37414 (cit. on pp. 12, 75).

Crockett, Thomas W. and Tobias Orloff (1993). 'A MIMD Rendering Algorithm for Distributed Memory Architectures'. In: *Proceedings of the 1993 Symposium on Parallel Rendering*. PRS '93. San Jose, California, USA: ACM, pp. 35–42. ISBN: 0-89791-618-2. DOI: 10.1145/166181.166186. URL: http://doi.acm.org/10.1145/166181.166186 (cit. on p. 109).

Dan Crişu (2012). 'Hardware algorithms for tile-based real-time rendering'. PhD thesis. Delft University of Technology. ISBN: 978-90-72298-26-3 (cit. on p. 17).

Deering, M. F. (Sept. 1993). 'Data complexity for virtual reality: where do all the triangles go?' In: *Proceedings of IEEE Virtual Reality Annual International Symposium*, pp. 357–363 (cit. on p. 110).

Deering, Michael (1995). 'Geometry Compression'. In: *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '95. New York, NY, USA: ACM, pp. 13–20. ISBN: 0-89791-701-4. DOI: 10.1145/218380.218391. URL: http://doi.acm.org/10.1145/218380.218391 (cit. on p. 15).

Developer's Guide for Intel® Processor Graphics For 4th Generation Intel® Core™ Processors (2013). Manual. Intel Corporation (cit. on p. 87).

Eldridge, Matthew Willard (2001). 'Designing Graphics Architectures Around Scalability and Communication'. AAI3026802. PhD thesis. ISBN: 0-493-38235-6 (cit. on pp. 17, 120).

Eldridge, Matthew, Homan Igehy and Pat Hanrahan (2000). 'Pomegranate: A Fully Scalable Graphics Architecture'. In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., pp. 443–454. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344981. URL: http://dx.doi.org/10.1145/344779.344981 (cit. on p. 17).

Elliott, Glenn A. and James H. Anderson (2012). 'Globally scheduled real-time multiprocessor systems with GPUs'. English. In: *Real-Time Systems* 48.1, pp. 34–74. ISSN: 0922-6443. DOI: 10.1007/s11241-011-9140-y. URL: http://dx.doi.org/10.1007/s11241-011-9140-y (cit. on p. 8).

Evans, Francine, Steven Skiena and Amitabh Varshney (1996). 'Optimizing Triangle Strips for Fast Rendering'. In: *Proceedings of the 7th Conference on Visualization '96*. VIS '96. San Francisco, California, USA: IEEE Computer Society Press, pp. 319–326. ISBN: 0-89791-864-9. URL: http://dl.acm.org/citation.cfm?id=244979.245626 (cit. on p. 15).

Forsyth, Tom (2006). *Linear-speed vertex cache optimisation*. URL: https://tomforsyth1000.github.io/papers/fast_vert_cache_opt.html (cit. on pp. 15, 86, 99).

Giacomoni, John, Tipp Moseley and Manish Vachharajani (2008). '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. Salt Lake City, UT, USA: ACM, pp. 43–52. ISBN: 978-1-59593-795-7. DOI: 10.1145/1345206.1345215. URL: http://doi.acm.org/10.1145/1345206.1345215 (cit. on p. 10).

Giesen, Fabian (2011). *A trip through the Graphics Pipeline 2011*. Retrieved: June 4th, 2018. URL: https://fgiesen.wordpress.com/2011/07/03/a-trip-through-the-graphics-pipeline-2011-part-3/ (cit. on p. 85).

Gottlieb, Allan, Boris D. Lubachevsky and Larry Rudolph (Apr. 1983). 'Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors'. In: *ACM Trans. Program. Lang. Syst.* 5.2, pp. 164–189. ISSN: 0164-0925. DOI: 10.1145/69624.357206. URL: http://doi.acm.org/10.1145/69624.357206 (cit. on pp. 11, 31).

Guo, Baining (1998). 'Progressive Radiance Evaluation Using Directional Coherence Maps'. In: *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '98. New York, NY, USA: ACM, pp. 255–266. ISBN: 0-89791-999-8. DOI: 10.1145/280814.280888. URL: http://doi.acm.org/10.1145/280814.280888 (cit. on p. 13).

Hachisuka, Toshiya, Wojciech Jarosz, Richard Peter Weistroffer, Kevin Dale, Greg Humphreys, Matthias Zwicker and Henrik Wann Jensen (Aug. 2008). 'Multidimensional Adaptive Sampling and Reconstruction for Ray Tracing'. In: *ACM Trans. Graph.* 27.3, 33:1–33:10. ISSN: 0730-0301. DOI: 10.1145/1360612.1360632. URL: http://doi.acm.org/10.1145/1360612.1360632 (cit. on pp. 4, 13, 14, 78).

Hargreaves, Shawn (2005). 'Generating shaders from HLSL fragments'. In: *ShaderX3: Advanced rendering with DirectX and OpenGL* (cit. on p. 9).

Harris, Mark (2014). 'Maxwell: The most advanced CUDA GPU ever made'. In: (cit. on p. 32).

Hendler, Danny, Itai Incze, Nir Shavit and Moran Tzafrir (2010). 'Flat combining and the synchronization-parallelism tradeoff'. In: *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*. SPAA '10. Thira, Santorini, Greece: ACM, pp. 355–364. ISBN: 978-1-4503-0079-7. DOI: 10.1145/1810479.1810540. URL: http://doi.acm.org/10.1145/1810479.1810540 (cit. on pp. 30, 31).

Hendler, Danny, Yossi Lev, Mark Moir and Nir Shavit (Feb. 2006). 'A Dynamic-sized Nonblocking Work Stealing Deque'. In: *Distrib. Comput.* 18.3, pp. 189–207. ISSN: 0178-2770. DOI: 10.1007/s00446-005-0144-5. URL: http://dx.doi.org/10.1007/s00446-005-0144-5 (cit. on pp. 10, 42).

Herlihy, Maurice P. and Jeannette M. Wing (July 1990). 'Linearizability: A Correctness Condition for Concurrent Objects'. In: *ACM Trans. Program. Lang. Syst.* 12.3, pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972. URL: http://doi.acm.org/10.1145/78969.78972 (cit. on pp. 10, 39).

Herlihy, Maurice, Victor Luchangco and Mark Moir (2003). 'Obstruction-Free Synchronization: Double-Ended Queues as an Example'. In: *Proceedings of the 23rd International Conference on Distributed Computing Systems*. ICDCS '03. Washington, DC, USA: IEEE Computer Society, pp. 522–. ISBN: 0-7695-1920-2. URL: http://dl.acm.org/citation.cfm?id=850929.851942 (cit. on pp. 30, 31).

Hoffman, Moshe, Ori Shalev and Nir Shavit (2007). 'The baskets queue'. In: *Proceedings of the 11th international conference on Principles of distributed systems*. OPODIS'07. Guadeloupe, French West Indies: Springer-Verlag, pp. 401–414. URL: http://dl.acm.org/citation.cfm?id=1782394.1782423 (cit. on pp. 11, 31).

Hoppe, Hugues (1999). 'Optimization of Mesh Locality for Transparent Vertex Caching'. In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., pp. 269–276. ISBN: 0-201-48560-5.

DOI: 10.1145/311535.311565. URL: http://dx.doi.org/10.1145/311535.311565 (cit. on pp. 15, 86, 94, 99, 101).

Hunt, Warren (2015). *Virtual Reality: The Next Great Graphics Revolution*. High Performance Graphics, 2015, Keynote (cit. on p. 52).

Isenburg, Martin and Peter Lindstrom (Nov. 2005). 'Streaming meshes'. In: *IEEE Visualization*, pp. 231–238. ISBN: 0-7803-9462-3 (cit. on p. 16).

Jia, Zhe, Marco Maggioni, Benjamin Staiger and Daniele Paolo Scarpazza (2018). 'Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking'. In: *CoRR* abs/1804.06826. arXiv: 1804.06826. URL: http://arxiv.org/abs/1804.06826 (cit. on p. 86).

Jones, Stephen (2012). 'Introduction to dynamic parallelism'. In: *GPU Technology Conference Presentation S*. Vol. 338 (cit. on p. 54).

Juliachs, M., T. Carrard and J.-P. Nominé (2007). 'Hybrid CPU-GPU Unstructured Meshes Parallel Volume Rendering on PC Clusters'. In: *Proceedings of the 7th Eurographics Conference on Parallel Graphics and Visualization*. EGPGV '07. Lugano, Switzerland: Eurographics Association, pp. 85–92. ISBN: 978-3-905673-50-0. DOI: 10.2312/EGPGV/EGPGV07/085-092. URL: http://dx.doi.org/10.2312/EGPGV/EGPGV07/085-092 (cit. on p. 17).

Karras, Tero (2012). 'Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees'. In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*. EGGH-HPG'12. Paris, France: Eurographics Association, pp. 33–37. ISBN: 978-3-905674-41-5. DOI: 10.2312/EGGH/HPG12/033-037. URL: http://dx.doi.org/10.2312/EGGH/HPG12/033-037 (cit. on p. 116).

Kato, Shinpei, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa and R. Rajkumar (Nov. 2011). 'RGEM: A Responsive GPGPU Execution Model for Runtime Engines'. In: *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pp. 57–66. DOI: 10.1109/RTSS.2011.13 (cit. on p. 8).

Kato, Shinpei, Karthik Lakshmanan, Raj Rajkumar and Yutaka Ishikawa (2011). 'TimeGraph: GPU scheduling for real-time multi-tasking environments'. In: *Proc. USENIX ATC*, pp. 17–30 (cit. on p. 8).

Kenzel, Michael, Bernhard Kerbl, Dieter Schmalstieg and Markus Steinberger (Nov. 2018). 'A High-Performance Software Graphics Pipeline Architecture for the GPU'. In: *ACM Trans. Graph.* 37.4 (cit. on pp. 24, 127).

Kenzel, Michael, Bernhard Kerbl, Wolfgang Tatzgern, Elena Ivanchenko, Dieter Schmalstieg and Markus Steinberger (Aug. 2018). 'On-the-fly Vertex Reuse for Massively-Parallel Software Geometry Processing'. In: *Proc. ACM Comput. Graph. Interact. Tech.* 1.2 (cit. on pp. 14, 28, 86).

Kerbl, Bernhard, Denis Kalkofen, Markus Steinberger and Dieter Schmalstieg (2015). 'Interactive Disassembly Planning for Complex Objects'. In: *Computer Graphics Forum* 34.2 (cit. on p. 27).

Kerbl, Bernhard, Michael Kenzel, Elena Ivanchenko, Dieter Schmalstieg and Markus Steinberger (Aug. 2018). 'Revisiting The Vertex Cache: Understanding and Optimizing Vertex Processing on the modern GPU'. In: *Proc. ACM Comput. Graph. Interact. Tech.* 1.2 (cit. on p. 26).

Kerbl, Bernhard, Michael Kenzel, Joerg H. Mueller, Dieter Schmalstieg and Markus Steinberger (2018). 'The Broker Queue: A Fast, Linearizable FIFO Queue for Fine-Granular Work Distribution on the GPU'. In: *Proceedings of the International Conference on Supercomputing*. ICS '18. Beijing, China (cit. on p. 20).

Kerbl, Bernhard, Michael Kenzel, Dieter Schmalstieg, Hans-Peter Seidel and Markus Steinberger (2016). 'Hierarchical Bucket Queuing for Fine-Grained Priority Scheduling on the GPU'. In: *Computer Graphics Forum* (cit. on p. 20).

Kerbl, Bernhard, Michael Kenzel, Dieter Schmalstieg and Markus Steinberger (2017). 'Effective Static Bin Patterns for Sort-middle Rendering'. In: *Proceedings of High Performance Graphics*. HPG '17. Los Angeles, California: ACM (cit. on p. 26).

Kerbl, Bernhard, Joerg H. Mueller, Michael Kenzel, Dieter Schmalstieg and Markus Steinberger (2018). 'A Scalable Queue for Work Distribution on GPUs'. In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '18. Vienna, Austria: ACM (cit. on p. 28).

Kerbl, Bernhard, Philip Voglreiter, Rostislav Khlebnikov, Dieter Schmalstieg, Daniel Seider, Michael Moche, Philipp Stiegler, R. Horst Portugaller and Bernhard Kainz (2013). 'Intervention Planning of Hepatocellular Carcinoma Radio-Frequency Ablations'. In: *Clinical Image-Based Procedures. From Planning to Intervention*. Vol. 7761. Lecture Notes in Computer Science. Springer Berlin Heidelberg (cit. on p. 27).

Kettunen, Markus, Marco Manzi, Miika Aittala, Jaakko Lehtinen, Frédo Durand and Matthias Zwicker (July 2015). 'Gradient-domain Path Tracing'. In: *ACM Trans. Graph.* 34.4, 123:1–123:13. ISSN: 0730-0301. DOI: 10.1145/2766997. URL: http://doi.acm.org/10.1145/2766997 (cit. on p. 14).

Khronos-Group (2015). *The OpenCL Specification 2.1* (cit. on p. 9).

Kubisch, Christoph (2015). *Life of a triangle – NVIDIA's logical pipeline*. Tech. rep. NVIDIA Corporation. URL: https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline (visited on 04/01/2017) (cit. on p. 86).

Kung, H. T. and John T. Robinson (June 1981). 'On optimistic methods for concurrency control'. In: *ACM Trans. Database Syst.* 6.2, pp. 213–226. ISSN: 0362-5915. DOI: 10.1145/319566.319567. URL: http://doi.acm.org/10.1145/319566.319567 (cit. on pp. 30, 31).

Laine, Samuli and Tero Karras (2011). 'High-performance Software Rasterization on GPUs'. In: *Proc. High Performance Graphics*. HPG '11, pp. 79–88. ISBN: 978-1-4503-0896-0 (cit. on p. 16).

Laine, Samuli, Tero Karras and Timo Aila (2013). 'Megakernels Considered Harmful: Wavefront Path Tracing on GPUs'. In: *Proc. High-Performance Graphics*. HPG '13. Anaheim, California: ACM, pp. 137–143. ISBN: 978-1-4503-2135-8. DOI: 10.1145/2492045.2492060. URL: http://doi.acm.org/10.1145/2492045.2492060 (cit. on p. 9).

Lamport, Leslie (Apr. 1983). 'Specifying Concurrent Program Modules'. In: *ACM Trans. Program. Lang. Syst.* 5.2, pp. 190–222. ISSN: 0164-0925. DOI: 10.1145/69624.357207. URL: http://doi.acm.org/10.1145/69624.357207 (cit. on p. 10).

Lee, Haeseung and M.A. Al Faruque (Mar. 2014). 'GPU-EvR: Run-time event based real-time scheduling framework on GPGPU platform'. In: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pp. 1–6. DOI: 10.7873/DATE.2014.233 (cit. on p. 8).

Lee, Patrick P. C., Tian Bu and Girish Chandranmenon (2009). '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. Princeton, New Jersey: ACM, pp. 78–79. ISBN: 978-1-60558-630-4. DOI: 10.1145/1882486.1882508. URL: http://doi.acm.org/10.1145/1882486.1882508 (cit. on p. 10).

Lehtinen, Jaakko, Timo Aila, Samuli Laine and Frédo Durand (July 2012). 'Reconstructing the Indirect Light Field for Global Illumination'. In: *ACM Trans. Graph.* 31.4, 51:1–51:10. ISSN: 0730-0301. DOI: 10.1145/2185520.2185547. URL: http://doi.acm.org/10.1145/2185520.2185547 (cit. on p. 14).

Leskovec, Jure and Andrej Krevl (June 2014). *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data (cit. on p. 47).

Li, Tzu-Mao, Yu-Ting Wu and Yung-Yu Chuang (Nov. 2012). 'SURE-based Optimization for Adaptive Sampling and Reconstruction'. In: *ACM Trans. Graph.* 31.6, 194:1–194:9. ISSN: 0730-0301. DOI: 10.1145/2366145.2366213. URL: http://doi.acm.org/10.1145/2366145.2366213 (cit. on p. 13).

Lin, G. and T. P. Y. Yu (July 2006). 'An improved vertex caching scheme for 3D mesh rendering'. In: *IEEE Transactions on Visualization and Computer*

Graphics 12.4, pp. 640–648. ISSN: 1077-2626. DOI: 10.1109/TVCG.2006.59 (cit. on pp. 15, 16, 86, 94, 99).

Liu, Xiao-Dan, Jia-Ze Wu and Chang-Wen Zheng (June 2012). 'KD-tree based parallel adaptive rendering'. In: *The Visual Computer* 28.6, pp. 613–623. ISSN: 1432-2315. DOI: 10.1007/s00371-012-0709-9. URL: https://doi.org/10.1007/s00371-012-0709-9 (cit. on p. 14).

Liu, Xiao-Dan and Chang-Wen Zheng (June 2013). 'Parallel adaptive sampling and reconstruction using multi-scale and directional analysis'. In: *The Visual Computer* 29.6, pp. 501–511. ISSN: 1432-2315. DOI: 10.1007/s00371-013-0814-4. URL: https://doi.org/10.1007/s00371-013-0814-4 (cit. on p. 14).

McManus, Donald and Carl Beckmann (1996). 'Optimal Static 2-dimensional Screen Subdivision for Parallel Rasterization Architectures'. In: *Proceedings of the Eleventh Eurographics Conference on Graphics Hardware*. EGGH'96. Poitiers, France: Eurographics Association, pp. 59–67. DOI: 10.2312/EGGH/EGGH96/059-067. URL: http://dx.doi.org/10.2312/EGGH/EGGH96/059-067 (cit. on p. 17).

Mehta, Soham Uday, JiaXian Yao, Ravi Ramamoorthi and Fredo Durand (July 2014). 'Factored Axis-aligned Filtering for Rendering Multiple Distribution Effects'. In: *ACM Trans. Graph.* 33.4, 57:1–57:12. ISSN: 0730-0301. DOI: 10.1145/2601097.2601113. URL: http://doi.acm.org/10.1145/2601097.2601113 (cit. on p. 14).

Membarth, Richard, Jan-Hugo Lupp, Frank Hannig, Jürgen Teich, Mario Körner and Wieland Eckert (2012). 'Dynamic Task-Scheduling and Resource Management for GPU Accelerators in Medical Imaging'. English. In: *Architecture of Computing Systems*. Ed. by Andreas Herkersdorf, Kay Römer and Uwe Brinkschulte. Vol. 7179. ARCS 2012. Springer Berlin Heidelberg, pp. 147–159. ISBN: 978-3-642-28292-8. DOI: 10.1007/978-3-642-28293-5_13. URL: http://dx.doi.org/10.1007/978-3-642-28293-5_13 (cit. on p. 8).

Michael, Maged M. and Michael L. Scott (1995). *Correction of a Memory Management Method for Lock-Free Data Structures*. Tech. rep. Rochester, NY, USA (cit. on p. 11).

Michael, Maged M. and Michael L. Scott (1996). '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. Philadelphia, Pennsylvania, USA: ACM, pp. 267–275. ISBN: 0-89791-800-2. DOI: 10.1145/248052.248106. URL: http://doi.acm.org/10.1145/248052.248106 (cit. on pp. 11, 31, 151).

Mitchell, Don P. (Aug. 1987). 'Generating Antialiased Images at Low Sampling Densities'. In: *SIGGRAPH Comput. Graph.* 21.4, pp. 65–72. ISSN: 0097-8930. DOI: 10.1145/37402.37410. URL: http://doi.acm.org/10.1145/37402.37410 (cit. on pp. 13, 52, 79).

Mohr, Peter, Bernhard Kerbl, Michael Donoser, Dieter Schmalstieg and Denis Kalkofen (2015). 'Retargeting Technical Documentation to Augmented Reality'. In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. CHI '15. Seoul, Republic of Korea: ACM (cit. on p. 27).

Molnar, Steven, Michael Cox, David Ellsworth and Henry Fuchs (July 1994). 'A Sorting Classification of Parallel Rendering'. In: *IEEE Comput. Graph. Appl.* 14.4, pp. 23–32. ISSN: 0272-1716. DOI: 10.1109/38.291528. URL: http://dx.doi.org/10.1109/38.291528 (cit. on pp. 17, 106).

Molnar, Steven, John Eyles and John Poulton (July 1992). 'PixelFlow: High-speed Rendering Using Image Composition'. In: *SIGGRAPH Comput. Graph.* 26.2, pp. 231–240. ISSN: 0097-8930. DOI: 10.1145/142920.134067. URL: http://doi.acm.org/10.1145/142920.134067 (cit. on p. 16).

Moon, Bochang, Nathan Carr and Sung-Eui Yoon (Sept. 2014). 'Adaptive Rendering Based on Weighted Local Regression'. In: *ACM Trans. Graph.* 33.5, 170:1–170:14. ISSN: 0730-0301. DOI: 10.1145/2641762. URL: http://doi.acm.org/10.1145/2641762 (cit. on p. 13).

Morrison, Adam and Yehuda Afek (Feb. 2013). 'Fast Concurrent Queues for x86 Processors'. In: *SIGPLAN Not.* 48.8, pp. 103–112. ISSN: 0362-1340. DOI: 10.1145/2517327.2442527. URL: http://doi.acm.org/10.1145/2517327.2442527 (cit. on pp. 12, 31, 38, 151).

Nocentino, Anthony E. and Philip J. Rhodes (2010). 'Optimizing Memory Access on GPUs Using Morton Order Indexing'. In: *Proceedings of the 48th Annual Southeast Regional Conference*. ACM SE '10. Oxford, Mississippi: ACM, 18:1–18:4. ISBN: 978-1-4503-0064-3. DOI: 10.1145/1900008.1900035. URL: http://doi.acm.org/10.1145/1900008.1900035 (cit. on p. 116).

NVIDIA (2009). *Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Retrieved April 6, 2017 (cit. on pp. 107, 161).

Nvidia (2012). *Next generation CUDA computer architecture Kepler GK110* (cit. on p. 9).

Orozco, Daniel, Elkin Garcia, Rishi Khan, Kelly Livingston and Guang R. Gao (Jan. 2012). 'Toward High-throughput Algorithms on Many-core Architectures'. In: *ACM Trans. Archit. Code Optim.* 8.4, 49:1–49:21. ISSN:

1544-3566. DOI: 10.1145/2086696.2086728. URL: http://doi.acm.org/10.1145/2086696.2086728 (cit. on pp. 11, 31, 32).

Overbeck, Ryan S., Craig Donner and Ravi Ramamoorthi (Dec. 2009). 'Adaptive Wavelet Rendering'. In: *ACM Trans. Graph.* 28.5, 140:1–140:12. ISSN: 0730-0301. DOI: 10.1145/1618452.1618486. URL: http://doi.acm.org/10.1145/1618452.1618486 (cit. on pp. 4, 13, 14, 52, 78).

Parker, Steven G. et al. (July 2010). 'OptiX: a general purpose ray tracing engine'. In: *ACM Trans. Graph.* 29.4, 66:1–66:13. ISSN: 0730-0301. DOI: 10.1145/1778765.1778803. URL: http://doi.acm.org/10.1145/1778765.1778803 (cit. on p. 9).

Patney, Anjul, Mohamed S. Ebeida and John D. Owens (2009). 'Parallel View-dependent Tessellation of Catmull-Clark Subdivision Surfaces'. In: *Proceedings of the Conference on High Performance Graphics 2009*. HPG '09. New Orleans, Louisiana: ACM, pp. 99–108. ISBN: 978-1-60558-603-8. DOI: 10.1145/1572769.1572785. URL: http://doi.acm.org/10.1145/1572769.1572785 (cit. on p. 13).

Patney, Anjul and John D. Owens (Dec. 2008). 'Real-time Reyes-style Adaptive Surface Subdivision'. In: *ACM Trans. Graph.* 27.5, 143:1–143:8. ISSN: 0730-0301. DOI: 10.1145/1409060.1409096. URL: http://doi.acm.org/10.1145/1409060.1409096 (cit. on p. 13).

Patney, Anjul, Stanley Tzeng, Kerry A. Seitz Jr. and John D. Owens (July 2015). 'Piko: A Framework for Authoring Programmable Graphics Pipelines'. In: *ACM Trans. Graph.* 34.4, 147:1–147:13. ISSN: 0730-0301. DOI: 10.1145/2766973. URL: http://doi.acm.org/10.1145/2766973 (cit. on p. 16).

Purcell, Tim (2010). 'Fast Tessellated Rendering on the Fermi GF100'. In: *High Performance Graphics Conf., Hot 3D presentation* (cit. on pp. 86, 107, 127).

Riguer, Guennadi (2006). *The Radeon X1000 Series Programming Guide* (cit. on p. 15).

Rousselle, Fabrice, Claude Knaus and Matthias Zwicker (Dec. 2011). 'Adaptive Sampling and Reconstruction Using Greedy Error Minimization'. In: *ACM Trans. Graph.* 30.6, 159:1–159:12. ISSN: 0730-0301. DOI: 10.1145/2070781.2024193. URL: http://doi.acm.org/10.1145/2070781.2024193 (cit. on pp. 4, 13, 52).

Sander, Pedro V., Diego Nehab and Joshua Barczak (July 2007). 'Fast Triangle Reordering for Vertex Locality and Reduced Overdraw'. In: *ACM Trans. Graph.* 26.3. ISSN: 0730-0301. DOI: 10.1145/1276377.1276489. URL: http://doi.acm.org/10.1145/1276377.1276489 (cit. on pp. 15, 86, 99).

Satish, Nadathur, Mark Harris and Michael Garland (2009). 'Designing efficient sorting algorithms for manycore GPUs'. In: *Proc. International Symposium on Parallel&Distributed Processing*. IPDPS '09. Washington, DC,

USA: IEEE Computer Society, pp. 1–10. ISBN: 978-1-4244-3751-1. DOI: 10.1109/IPDPS.2009.5161005. URL: http://dx.doi.org/10.1109/IPDPS.2009.5161005 (cit. on p. 9).

Sattlecker, Martin and Markus Steinberger (2015). 'Reyes Rendering on the GPU'. In: *Proceedings of the 31st Spring Conference on Computer Graphics*. SCCG '15. Smolenice, Slovakia: ACM, pp. 31–38. ISBN: 978-1-4503-3693-2. DOI: 10.1145/2788539.2788543. URL: http://doi.acm.org/10.1145/2788539.2788543 (cit. on p. 13).

Scogland, Thomas R.W. and Wu-chun Feng (2015). 'Design and Evaluation of Scalable Concurrent Queues for Many-Core Architectures'. In: *Proc. ACM/SPEC International Conference on Performance Engineering*. ICPE '15. Austin, Texas, USA: ACM, pp. 63–74. ISBN: 978-1-4503-3248-4. DOI: 10.1145/2668930.2688048. URL: http://doi.acm.org/10.1145/2668930.2688048 (cit. on pp. 12, 30, 31).

Seiler, Larry et al. (Aug. 2008). 'Larrabee: A Many-core x86 Architecture for Visual Computing'. In: *ACM Trans. Graph.* 27.3, 18:1–18:15. ISSN: 0730-0301. DOI: 10.1145/1360612.1360617. URL: http://doi.acm.org/10.1145/1360612.1360617 (cit. on p. 16).

Shann, Chien-Hua, T.-L. Huang and Cheng Chen (2000). 'A practical nonblocking queue algorithm using compare-and-swap'. In: *Parallel and Distributed Systems, 2000. Proceedings. Seventh International Conference on*, pp. 470–475. DOI: 10.1109/ICPADS.2000.857731 (cit. on pp. 11, 31).

Sheaffer, J. W., D. Luebke and K. Skadron (2004). 'A Flexible Simulation Framework for Graphics Architectures'. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '04. Grenoble, France: ACM, pp. 85–94. ISBN: 3-905673-15-0. DOI: 10.1145/1058129.1058142. URL: http://doi.acm.org/10.1145/1058129.1058142 (cit. on p. 15).

Steinberger, Markus, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel and Dieter Schmalstieg (Nov. 2012). 'Softshell: dynamic scheduling on GPUs'. In: *ACM Trans. Graph.* 31.6, 161:1–161:11 (cit. on pp. 9, 19, 30, 55, 61).

Steinberger, Markus, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter and Dieter Schmalstieg (Nov. 2014). 'Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU'. In: *ACM Trans. Graph.* 33.6 (cit. on pp. 9, 13, 19, 30, 54, 55, 57, 75, 76).

Steinberger, Markus, Michael Kenzel, Bernhard Kainz and Dieter Schmalstieg (2012). 'ScatterAlloc: Massively parallel dynamic memory allocation for the GPU'. In: *Innovative Parallel Computing (InPar), 2012*, pp. 1–10. DOI: 10.1109/InPar.2012.6339604 (cit. on p. 32).

Tchiboukdjian, Marc, Vincent Danjean and Bruno Raffin (June 2008). 'A Fast Cache-Oblivious Mesh Layout with Theoretical Guarantees'. In: *International Workshop on Super Visualization (IWSV'o8)*. Kos, Greece. URL: https://hal.inria.fr/inria-00436053 (cit. on p. 16).

Tchiboukdjian, Marc, Vincent Danjean and Bruno Raffin (Sept. 2010). 'Binary Mesh Partitioning for Cache-Efficient Visualization'. In: *IEEE Transactions on Visualization and Computer Graphics* 16.5, pp. 815–828. ISSN: 1077-2626. DOI: 10.1109/TVCG.2010.19 (cit. on p. 16).

Tsigas, Philippas and Yi Zhang (2001). '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. Crete Island, Greece: ACM, pp. 134–143. ISBN: 1-58113-409-6. DOI: 10.1145/378580.378611. URL: http://doi.acm.org/10.1145/378580.378611 (cit. on pp. 11, 31).

Tzeng, Stanley, Anjul Patney and John D. Owens (2010). 'Task management for irregular-parallel workloads on the GPU'. In: *Proc. High Performance Graphics*. HPG '10. Saarbrucken, Germany: Eurographics Association, pp. 29–37. URL: http://dl.acm.org/citation.cfm?id=1921479.1921485 (cit. on pp. 9, 75).

Valois, John D. (1994). 'Implementing Lock-Free Queues'. In: *In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV*, pp. 64–69 (cit. on p. 11).

Vinkler, M., V. Havran, J. Bittner and J. Sochor (2015). 'Parallel On-Demand Hierarchy Construction on Contemporary GPUs'. In: *IEEE Transactions on Visualization and Computer Graphics* PP.99, pp. 1–1. ISSN: 1077-2626. DOI: 10.1109/TVCG.2015.2465898 (cit. on p. 9).

Wang, Po-Han, Chia-Lin Yang, Yen-Ming Chen and Yu-Jung Cheng (Oct. 2011). 'Power Gating Strategies on GPUs'. In: *ACM Trans. Archit. Code Optim.* 8.3, 13:1–13:25. ISSN: 1544-3566. DOI: 10.1145/2019608.2019612. URL: http://doi.acm.org/10.1145/2019608.2019612 (cit. on p. 15).

Wang, Lizhe, Dan Chen, Ze Deng and Fang Huang (July 2011). 'Review: Large Scale Distributed Visualization on Computational Grids: A Review'. In: *Comput. Electr. Eng.* 37.4, pp. 403–416. ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2011.05.010. URL: http://dx.doi.org/10.1016/j.compeleceng.2011.05.010 (cit. on p. 17).

Ward, Gregory J., Francis M. Rubinstein and Robert D. Clear (June 1988). 'A Ray Tracing Solution for Diffuse Interreflection'. In: *SIGGRAPH Comput. Graph.* 22.4, pp. 85–92. ISSN: 0097-8930. DOI: 10.1145/378456.378490. URL: http://doi.acm.org/10.1145/378456.378490 (cit. on p. 13).

Wen, Yuan, Zheng Wang and M.F.P. O'Boyle (Dec. 2014). 'Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms'. In: *High Performance Computing (HiPC), 2014*, pp. 1–10. DOI: 10.1109/HiPC.2014.7116910 (cit. on p. 8).

Yang, Chaoran and John Mellor-Crummey (2016). 'A Wait-free Queue As Fast As Fetch-and-add'. In: *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '16. Barcelona, Spain: ACM, 16:1–16:13. ISBN: 978-1-4503-4092-2. DOI: 10.1145/2851141.2851168. URL: http://doi.acm.org/10.1145/2851141.2851168 (cit. on pp. 12, 31, 151).

Yoon, Sung-eui and Peter Lindstrom (Nov. 2007). 'Random-Accessible Compressed Triangle Meshes'. In: *IEEE Transactions on Visualization and Computer Graphics* 13.6, pp. 1536–1543. ISSN: 1077-2626. DOI: 10.1109/TVCG.2007.70585 (cit. on p. 16).

Zhou, Kun, Qiming Hou, Zhong Ren, Minmin Gong, Xin Sun and Baining Guo (Dec. 2009). 'RenderAnts: Interactive Reyes Rendering on GPUs'. In: *ACM Trans. Graph.* 28.5, 155:1–155:11. ISSN: 0730-0301. DOI: 10.1145/1618452.1618501. URL: http://doi.acm.org/10.1145/1618452.1618501 (cit. on p. 13).

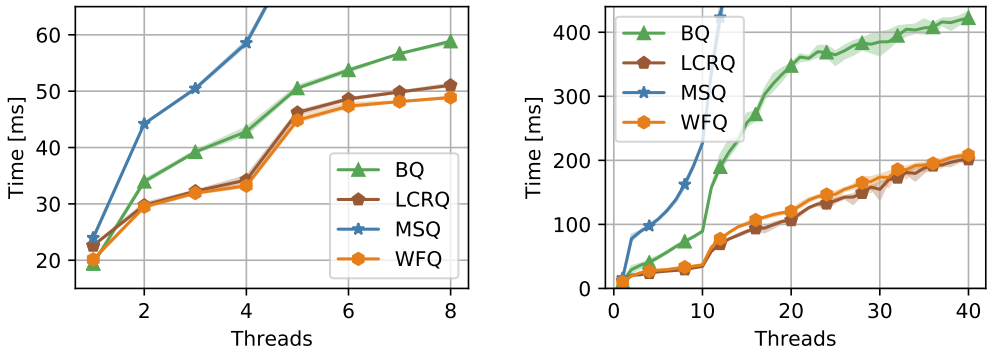
Zwicker, Matthias, Wojciech Jarosz, Jaakko Lehtinen, Bochang Moon, Ravi Ramamoorthi, Fabrice Rousselle, Pradeep Sen, Cyril Soler and Sungeui E. Yoon (2015). 'Recent Advances in Adaptive Sampling and Reconstruction for Monte Carlo Rendering'. In: *Computer Graphics Forum*. DOI: 10.1111/cgf.12592 (cit. on pp. 13, 14).

Appendix A.

Performance of the Broker Queue on Other Architectures

We revisit our argument regarding design choices in the broker queue, given that it targets GPU architectures. At the core of its functionality, the broker queue employs a Counter variable to weigh enqueued against dequeued elements in the queue. This counter is modified using atomic addition or subtraction primitives. Previous queue designs that target CPU architectures usually refrain from such designs, as they tend to create choke points. However, in our research, we have shown that the differences in design between CPU and GPU have significant influences on their respective performance for contended atomic operations. Hence, the broker queue exploits the peculiarities of the GPU architecture to achieve high performance, despite it employing a methodology that would be considered harmful on CPUs. Here we show that, on the CPU, the broker queue is easily bested by elaborate queuing techniques that were carefully tailored towards CPU architectures instead. Figures A.1a and A.1b plot runtimes for a queuing experiment, similar to the initial evaluation in Chapter 4 each thread performs a fixed number $N = 10\ 000\ 000$ of enqueue/dequeue pairs. We compare against the most recently presented techniques, LCRQ (Morrison and Afek, 2013) and WFQ (Yang and Mellor-Crummey, 2016), as well as the Michael-Scott queue (MSQ) (Michael and Scott, 1996). We use the readily available framework conceived for the evaluation of WFQ [1] and add an implementation of our own queuing algorithm, according to the provided pseudocode.

[1] Hosted at `https://github.com/chaoran/fast-wait-free-queue`.



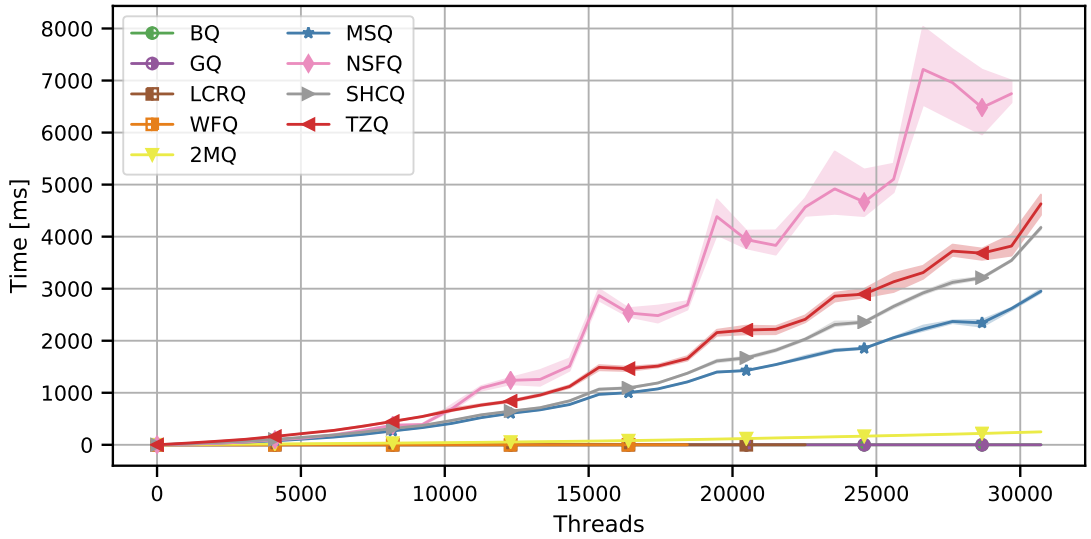
(a) Intel(R) Core(TM) i7-4820K CPU @3.70GHz

(b) Intel(R) Xeon(R) CPU E5-2680 v2 @2.80GHz

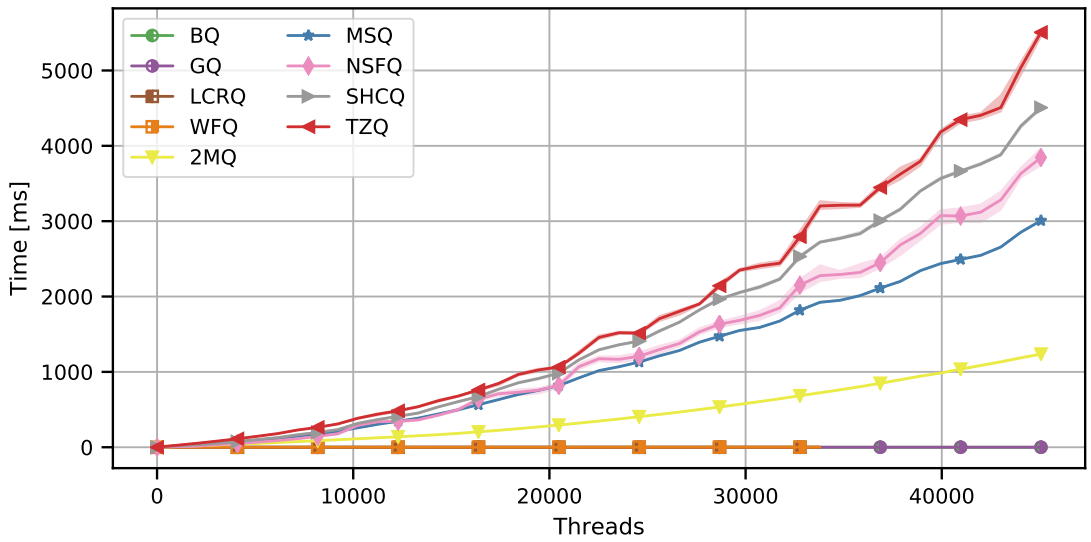
Figure A.1.: Evaluation of the broker queue on CPUs.

As expected, the sophisticated designs of LCRQ and WFQ outperform our approach on CPU architectures (see Figures A.1a and A.1b). In our experiment, the non-linear growth in runtime peaks at the maximum number of available threads at more than 2× compared to LCRQ or WFQ. Contention is easily identified to be the bottleneck and is even visible in the graphs. Figure A.1a clearly shows our approach to be the fastest at a single thread, due to its low code complexity. However, as the number of threads increase, this fact is reversed, and the broker queue quickly trails behind more complex, less contention-heavy alternatives. Figure A.1b show an increased runtime spike in the range from 10 to 20 threads. Given that each CPU on the target machine contains 10 cores, it is reasonable to assume that in this range, the requirement for contended atomic operations and implied synchronization between physical CPUs is extremely costly. In comparison, both WFQ and LCRQ experience only a slight uptick in total runtime. Interestingly, the broker queue is still significantly more efficient than the Michael-Scott queue. In conclusion, our experiments show that the broker queue is a poor choice and that fast alternatives exist for execution on the CPU. However, it also highlights how seemingly simple approaches can outperform established methods on GPUs: the weaknesses of the broker queue on the CPU (*i.e.*, relying on contended atomic addition) become its strengths on GPU hardware.

We ran the same experiments that were included in our full evaluation for two older Nvidia GPU generations. In addition to the original 1080Ti model (Pascal), we investigate GTX 780Ti (Kepler) and 980Ti (Maxwell) performance. Experiments for initial runtime, imbalanced workload and practical application (Page Rank) were run on each. The overall trends from Chapter 4 are confirmed by the following figures. We discuss results, deviations in performance and other artifacts in the accompanying captions below.

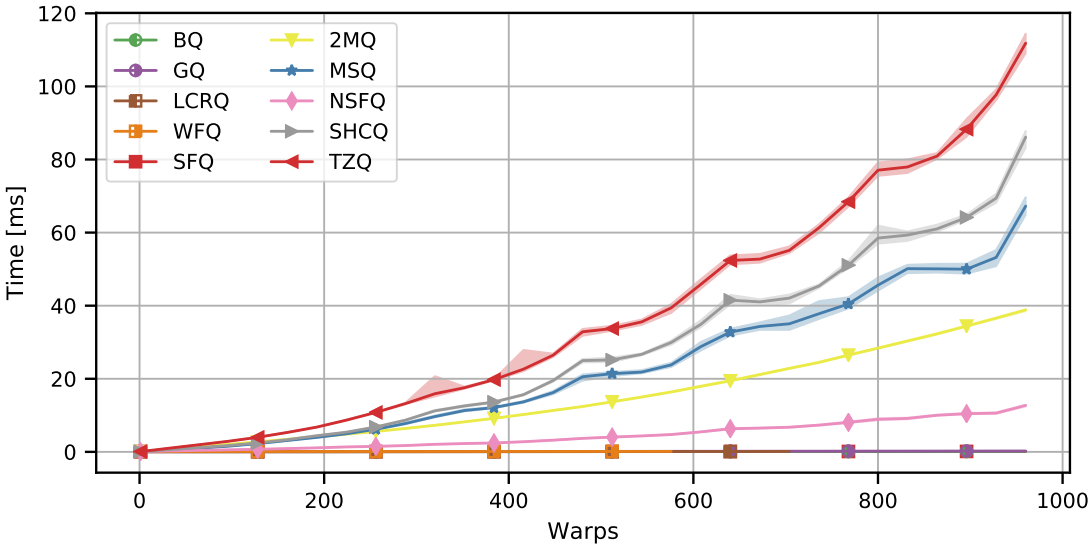


(a) Runtime of all queues, with thread granularity on *GTX 780Ti*

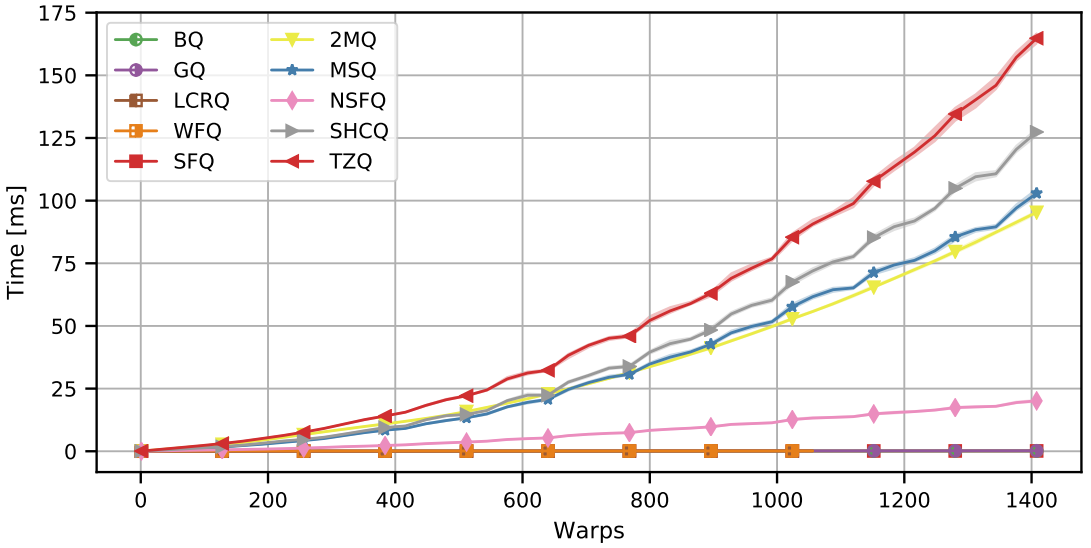


(b) Runtime of all queues, with thread granularity on *GTX 980Ti*

Figure A.2.: Running initial runtime comparison (10 enqueue/dequeue pairs) on all tested queues. As with our previously reported results, we can clearly distinguish two classes in terms of performance. The fastest parallel queuing algorithms in this basic case, thus warranting detailed assessment, are the LCRQ, WFQ, Gottlieb's Queue (GQ) and ours (BQ). The non-blocking version of the queue by Scogland-Feng is particularly slow on the 780Ti and less so on the 980Ti. In contrast, while still slower than the top contenders, the dual-mutex queue (2MQ) seems to perform much better on the Kepler architecture than on newer GPUs.

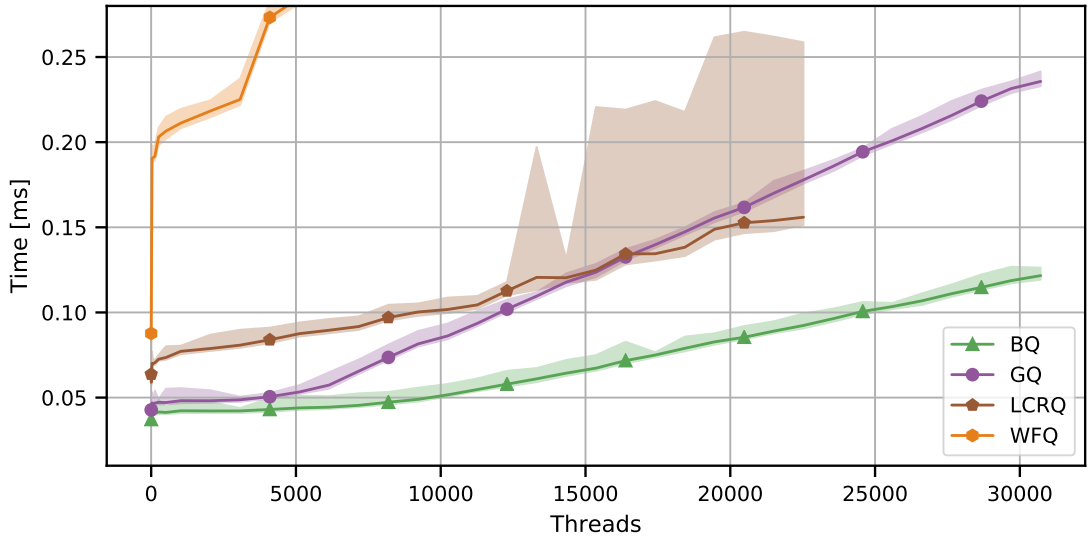


(a) Runtime of all queues, with warp granularity on *GTX 780Ti*

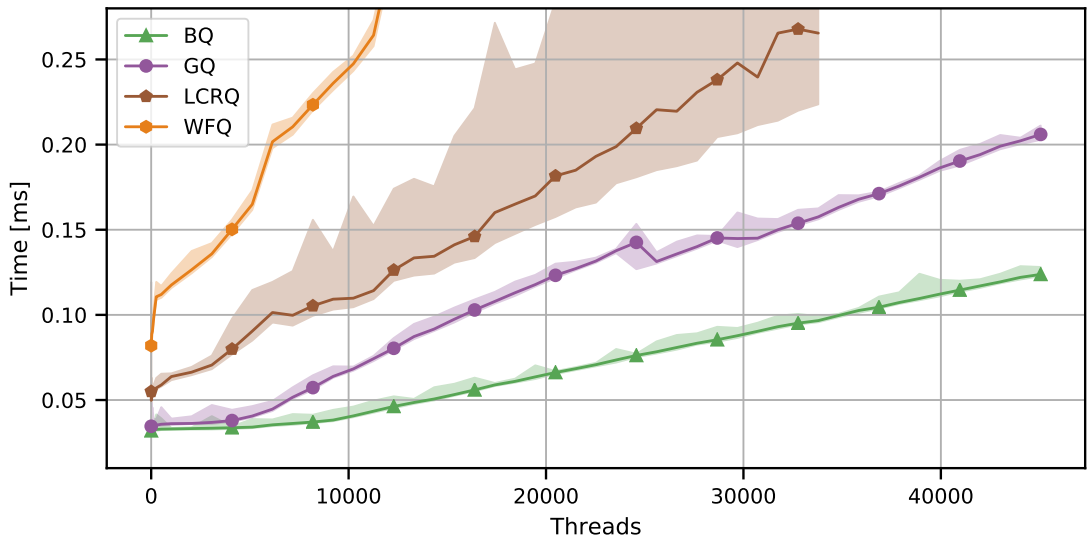


(b) Runtime of all queues, with warp granularity on *GTX 980Ti*

Figure A.3.: For per-warp execution, Scogland-Feng's fully blocking queue (SFQ) joins the contenders for highest performance. As the GTX 780Ti is the oldest and weakest model we tested, the restrictions for maximum occupancy are more severe. Hence, due to high code complexity, WFQ can only obtain 60% of the occupancy that simpler designs can achieve. Apart from that, all previous trends are confirmed, with queuing algorithms still clearly split into two groups in terms of sheer performance.

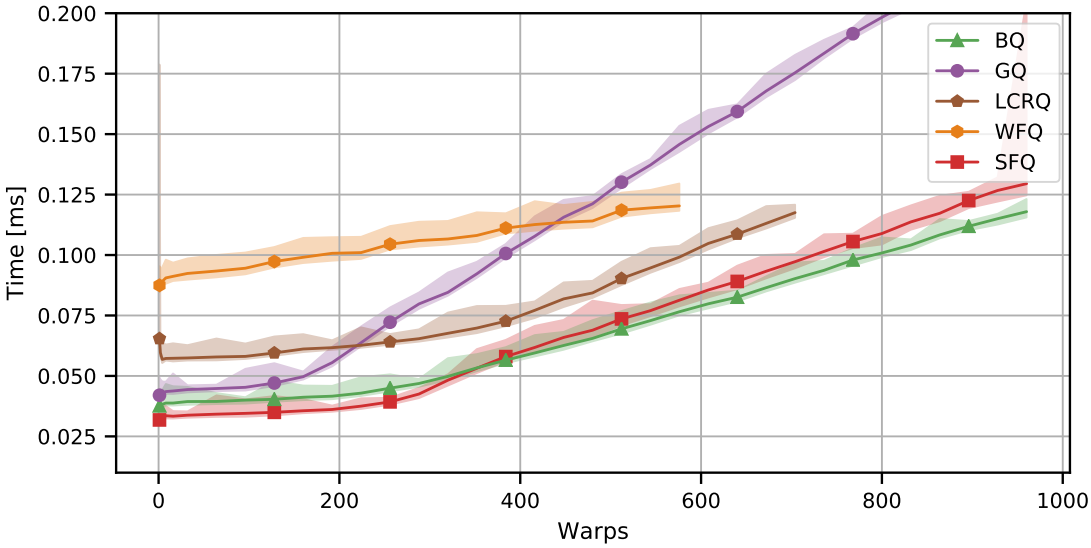


(a) Details for fastest queues, with thread granularity on *GTX 780Ti*

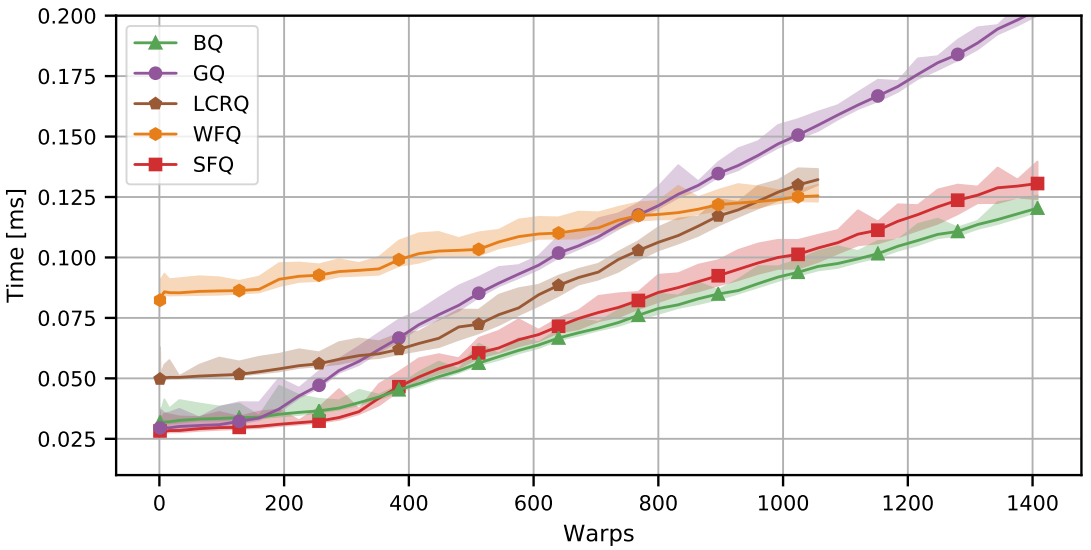


(b) Details for fastest queues, with thread granularity on *GTX 980Ti*

Figure A.4.: Detailed assessment of initial runtime comparison with per-thread granularity for the fastest queuing contenders. On the GTX 780Ti, the LCRQ surpasses the GQ towards the higher end of its spectrum of possible configurations, which differs from our previous results. Notice, however, that LCRQ maintains its high variance in runtime over 100 iterations, thus its worst performance is still higher than that of GQ. On the GTX 980Ti, relative performance is highly similar to reported results for Pascal, with LCRQ performing clearly worse than GQ and BQ. The slowest algorithm on both architectures remains the WFQ.

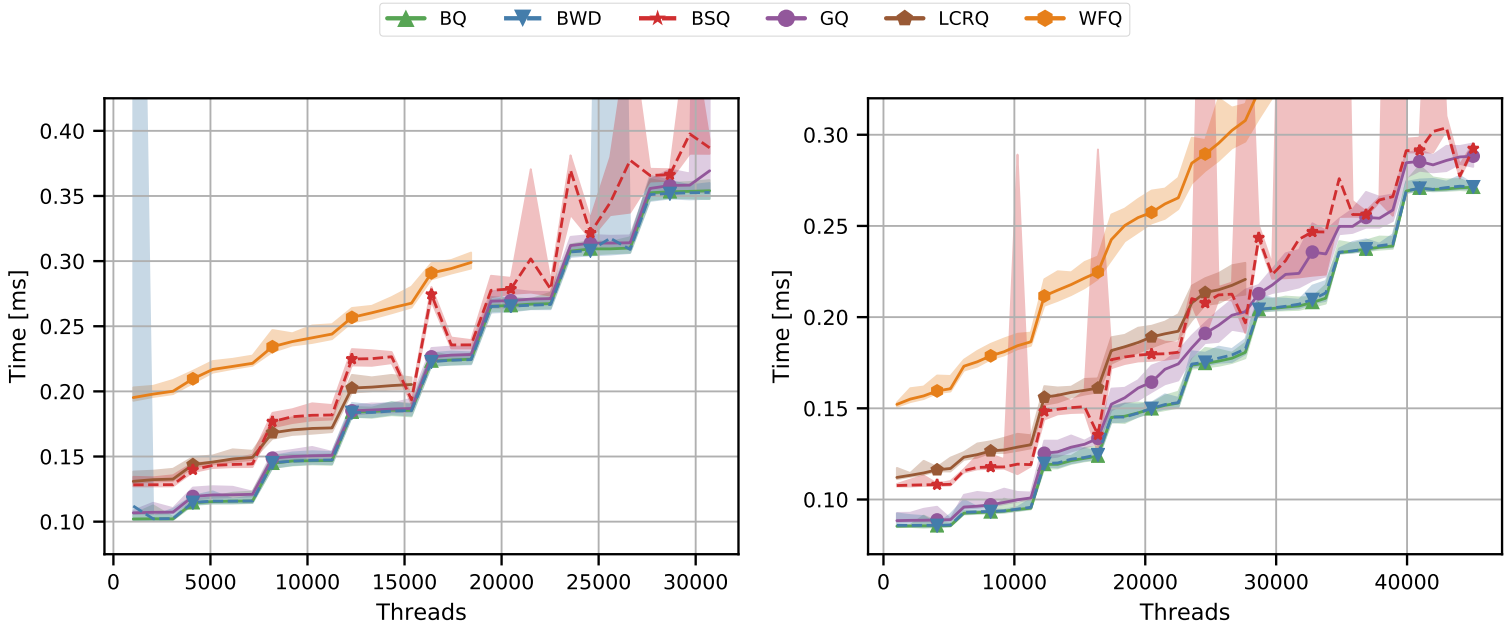


(a) Details for fastest queues, with warp granularity on *GTX 780Ti*



(b) Details for fastest queues, with warp granularity on *GTX 980Ti*

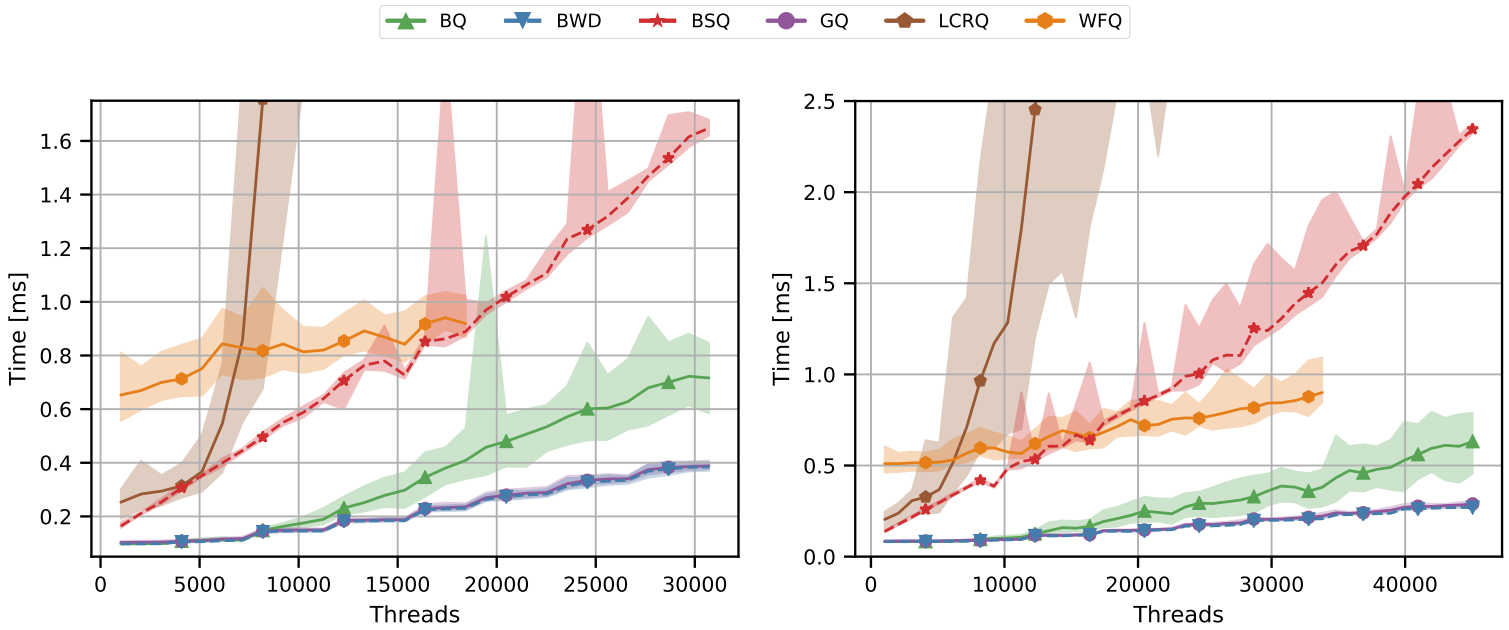
Figure A.5.: Details for fastest queues, with warp granularity. Due to higher occupancy granted by architecture specifications, the WFQ eventually bests LCRQ on the 980Ti. As the number of threads/warps in use increases, the scalable broker queue outperforms all other contenders. On the GTX 780Ti, it is up to 6% faster than even the fully blocking SFQ, which may only run in balanced scenarios with per-warp granularity. On the GTX 980Ti, this figure jumps to 12% advantage of BQ over SFQ.



(a) $\mathbb{P}(enq) = 50\%, \mathbb{P}(deq) = 25\%$ on *GTX 780Ti*

(b) $\mathbb{P}(enq) = 50\%, \mathbb{P}(deq) = 25\%$ on *GTX 980Ti*

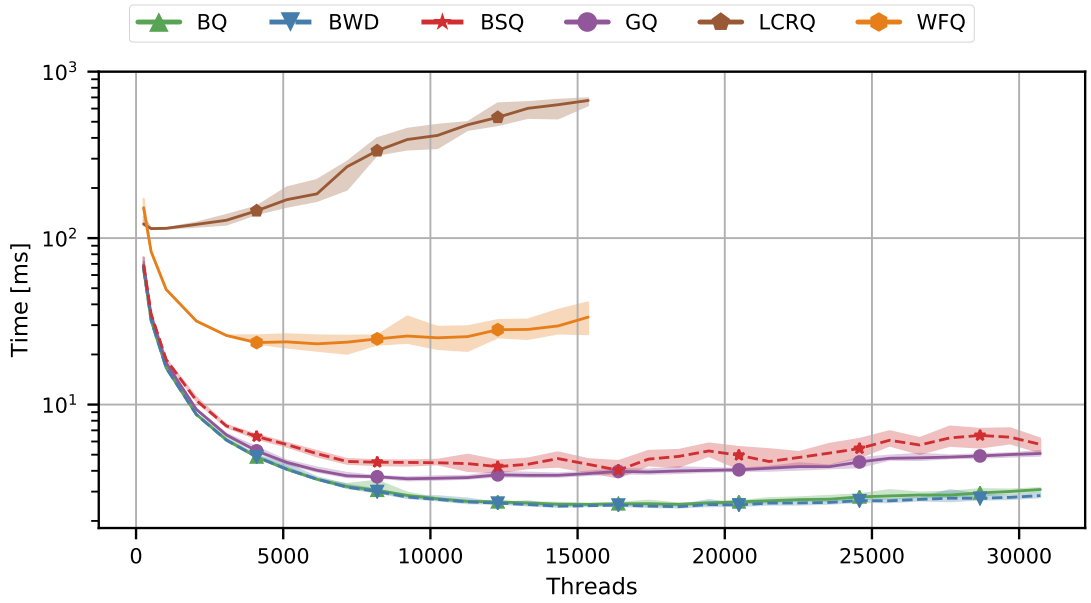
Figure A.6.: We test an imbalanced scenario where threads are more likely to enqueue than to dequeue. $\mathbb{P}(enq)$ and $\mathbb{P}(deq)$ denote probabilities with which threads perform enqueue and dequeue operations, respectively. Furthermore, we simulate a light workload by performing 128 fused multiply-adds (FMAs) between dequeue and enqueue. We always start with an empty queue and increase the number of threads that concurrently perform operations on it. On the 780Ti, work stealing (BSQ) appears to be slightly less effective than on the 980Ti, compared to other techniques. We also noticed fluctuations in runtime and hence an increase in variance of BWD, GQ and especially BSQ, which however does not affect their mean performance significantly.



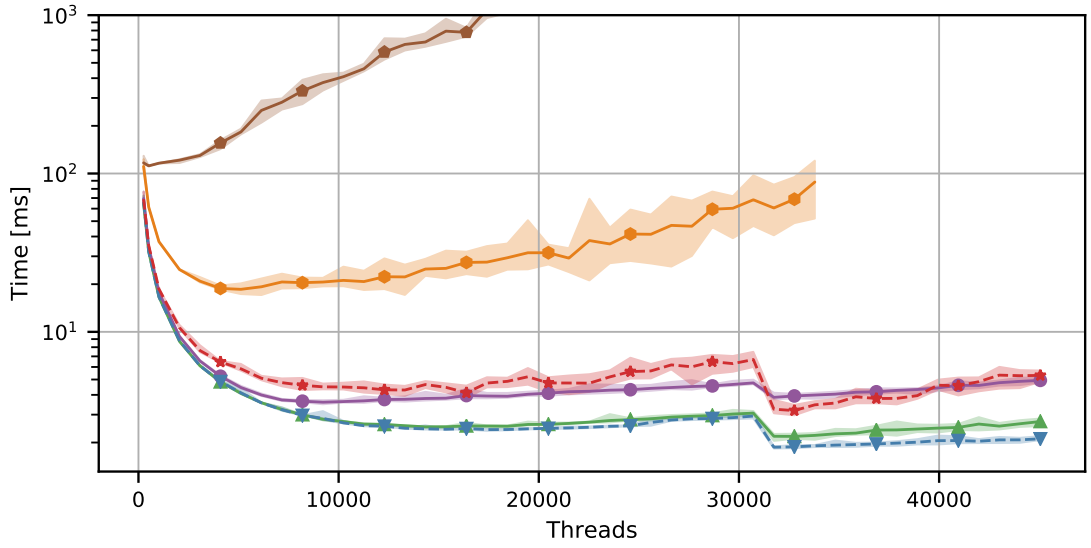
(a) $\mathbb{P}(enq) = 25\%, \mathbb{P}(deq) = 50\%$ on 780Ti

(b) $\mathbb{P}(enq) = 25\%, \mathbb{P}(deq) = 50\%$ on 980Ti

Figure A.7.: Testing the inverse of the previously described scenario, we have a probability for dequeuing higher than that for enqueuing. Consequently, we quickly hit a border case for the queuing algorithm, namely the **Empty** state. As more entries are consumed than are being emplaced, the queue needs to frequently handle threads failing to dequeue. While WFQ remains fairly stable thanks to its slow-path/fast-path mechanism, the LCRQ quickly starts to deteriorate as it is forced to continuously close and allocate new ring segments. The BQ is also affected by failing dequeues, but to a much lesser degree. The BSQ, whose work stealing scheme requires confirming that no more work is available in any queue, naturally gets consistently slower with an increasing number of threads, as more threads also imply more queues that need to be checked. On both Kepler and Maxwell architectures, BQ is the linearizable queue that is the least affected by frequent **Empty** states. The non-linearizable BWD and GQ are affected even less and—in contrast to results reported for the 1080Ti—are virtually tied for best performance.

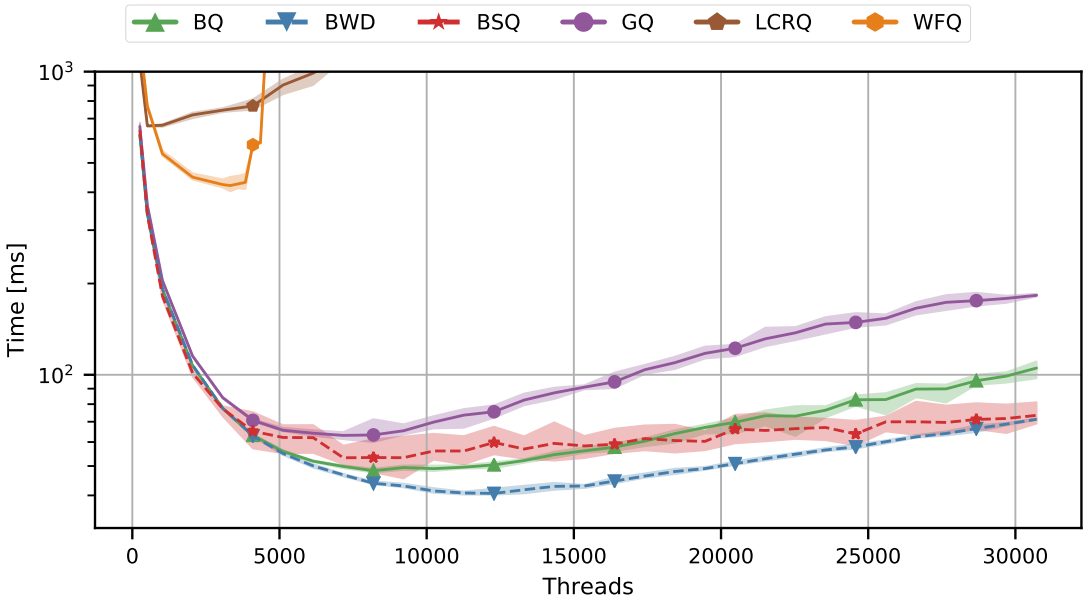


(a) Page Rank for *p2p-Gnutella31* on *GTX 780Ti*

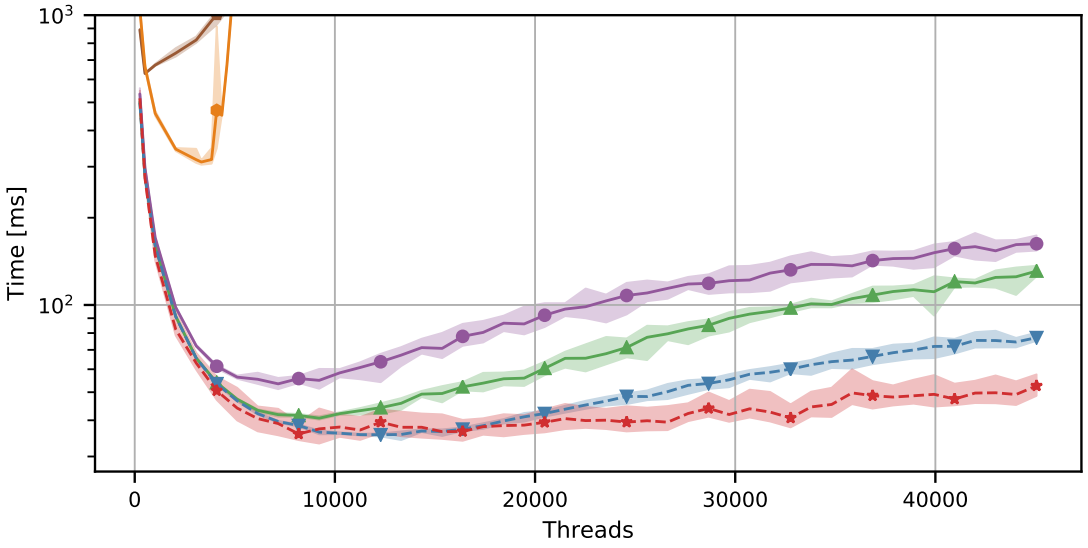


(b) Page Rank for *p2p-Gnutella31* on *GTX 980Ti*

Figure A.8.: We run our implementation of queuing-based Page Rank on Kepler and Maxwell architectures, using the *p2p-Gnutella31* network. The code complexity of the Page Rank procedure, on top of the underlying queuing algorithm, restricts potential occupancy even further. Note that on the GTX 780Ti, LCRQ and WFQ can achieve no more than 50% of the occupancy that is obtained with the remaining techniques. As the number of threads increases, the benefit of BQ and BWD over GQ rises up to 15–17%, respectively, on both architectures. BSQ is clearly trailing on the GTX 780Ti, but eventually on par with GQ on the GTX 980Ti, which is fitted with a higher number of compute cores and thus allows for an even higher level of parallelism.



(a) Page Rank for *web-NotreDame* on *GTX 780Ti*



(b) Page Rank for *web-NotreDame* on *GTX 980Ti*

Figure A.9.: We again consider the fastest identified parallel queuing algorithms for computing Page Rank values. The significantly more extensive input from the *web-NotreDame* test set contains 300k nodes and 1.5m edges. On both the 780Ti and the 980Ti, LCRQ and WFQ are quickly afflicted by excessively long runtime. The simpler GQ shows much better performance than the non-blocking queues but is still slower than all of our techniques. The results on the 980Ti closely mimic those for the 1080Ti, with BWD being faster than BQ due to reduced overhead, and BSQ benefitting from effective work stealing. However, on the 780Ti—as was already suggested above—work stealing appears to be less effective compared to a fast, centralized queuing approach. Hence, the BWD technique beats BSQ for our extensive Page Rank test case on the Kepler generation model.

Appendix B.

Identified GPU Binning Patterns

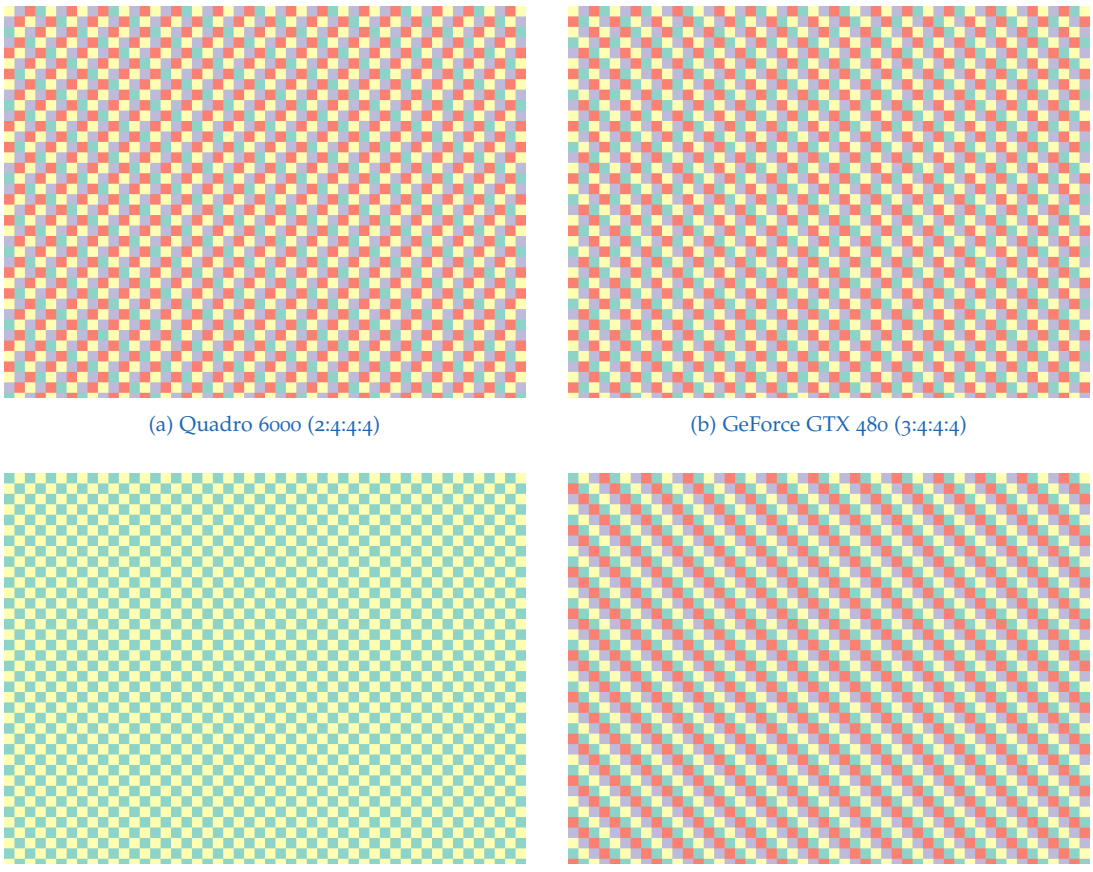
While each major GPU manufacturer publishes general information on their GPU architectures, details on how the hardware graphics pipeline is implemented are scarce to come by. To get an idea of how current GPU architectures distribute the rendering workload to their hardware rasterizers and/or programmable cores, we conducted a number of experiments designed to identify screen regions which share computational resources. The full set of results for all the GPUs we ran these experiments on is presented in this appendix.

On all recent Nvidia architectures, multiprocessors are grouped into graphics processing clusters (GPCs). Each GPC contains one rasterizer which has exclusive control over the GPC's multiprocessors (NVIDIA, 2009). For Nvidia's GPU models, we used the `NV_shader_thread_group` OpenGL extension which allows a fragment shader to query which multiprocessor the fragment is being processed on. Notice that less powerful models by Nvidia usually feature GPCs with imbalanced computing power, i.e., unequal SMs per GPC. This is indicated by the ";"-separated list following the GPC count and also reflected in the corresponding patterns (weaker GPCs occur less frequently).

For AMD and Intel models, we used an indirect measuring approach, as there is no similar support for querying executing processors in the fragment shader. Hence, we chose a large number N and instantiated N triangles (with Z-testing disabled) to cover exactly one pixel at a specific location p. Then, we iterated over different pixel locations p' and examined the effect of drawing N more triangles to cover this location as well. For those pixel locations p and p' where the time required to draw both sets of triangles is particularly high, we can assume that they map to the same rasterization unit.

Thus, we can use timing measurement to identify image regions that are assigned to the same rasterizer, or, at least, cause the GPU to behave as if that were the case. All identified patterns were reproducible in multiple launches and constant in their layout and bin size.

B.1. Nvidia Fermi



(a) Quadro 6000 (2:4:4:4)

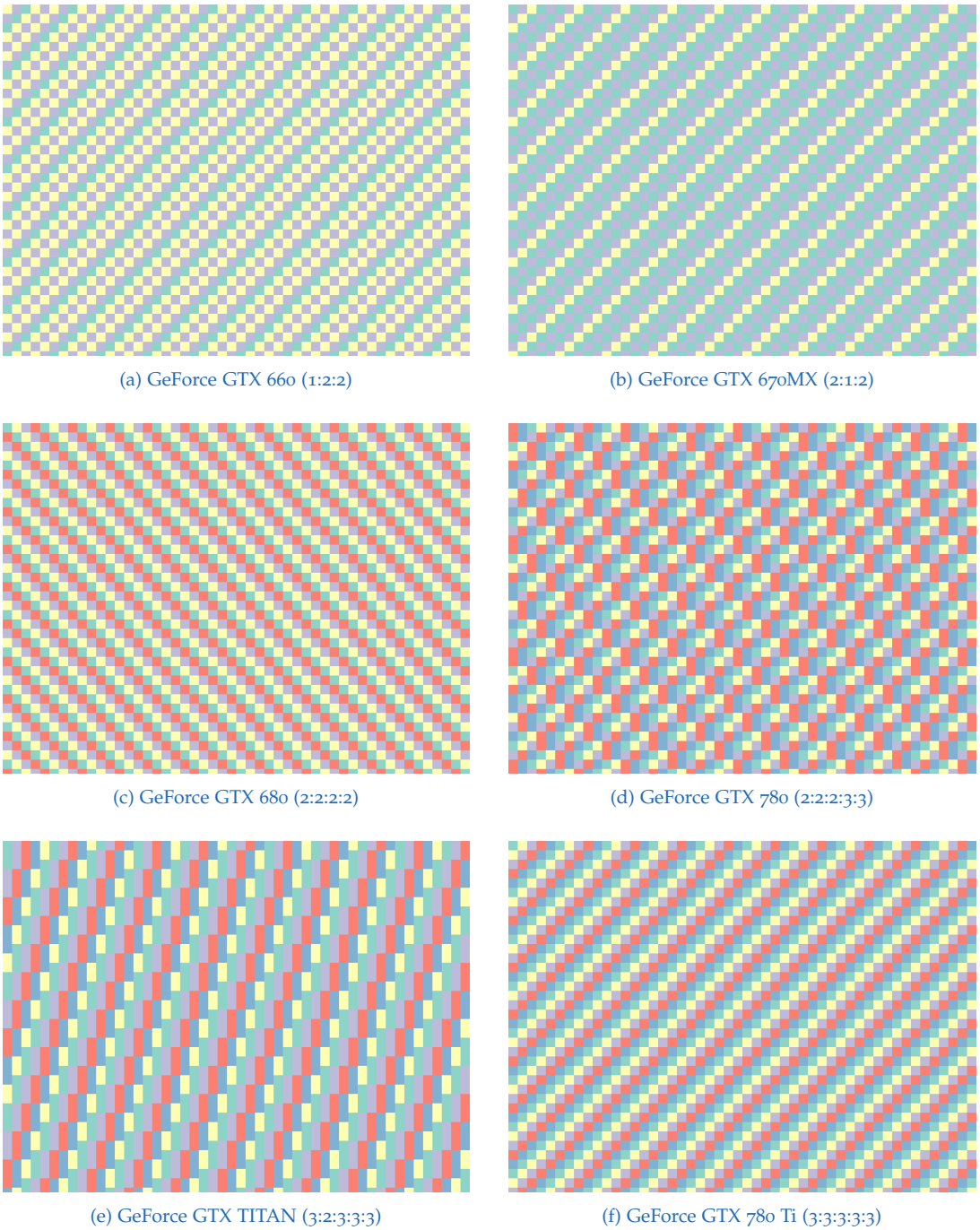
(b) GeForce GTX 480 (3:4:4:4)

(c) GeForce GTX 560Ti (4:4)

(d) GeForce GTX 580 (4:4:4:4)

Figure B.1.: Patterns used by the Nvidia Fermi architecture. Numbers in parentheses give the number of multiprocessors of the individual GPCs.

B.2. Nvidia Kepler



(a) GeForce GTX 660 (1:2:2)

(b) GeForce GTX 670MX (2:1:2)

(c) GeForce GTX 680 (2:2:2:2)

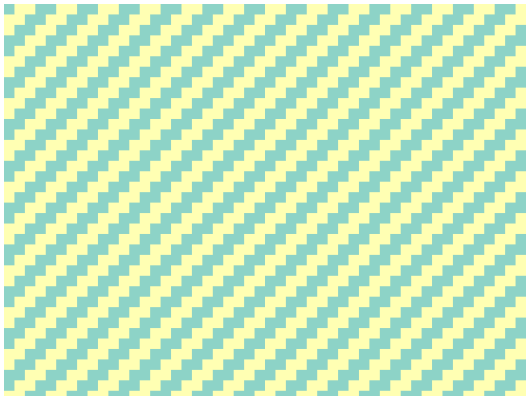
(d) GeForce GTX 780 (2:2:2:3:3)

(e) GeForce GTX TITAN (3:2:3:3:3)

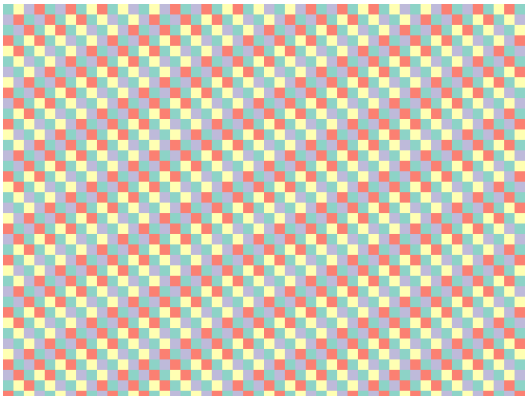
(f) GeForce GTX 780 Ti (3:3:3:3:3)

Figure B.2.: Binning patterns used by the Nvidia Kepler architecture.

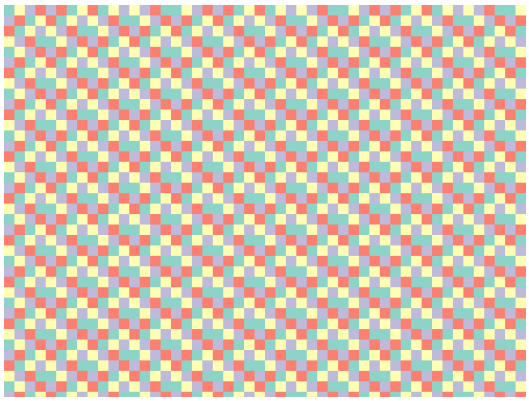
B.3. Nvidia Maxwell



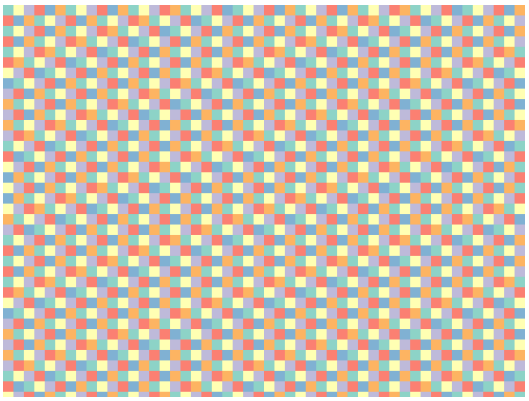
(a) GeForce GTX 960 (4:4)



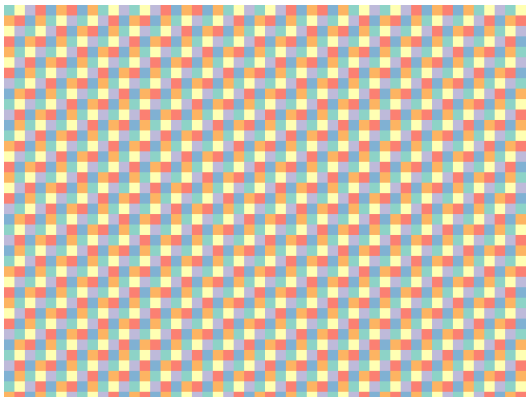
(b) GeForce GTX 970 (4:3:3:3)



(c) GeForce GTX 980 (4:4:4:4)



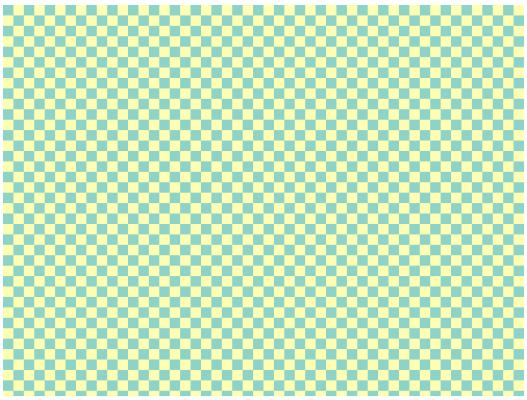
(d) GeForce GTX 980 Ti (4:4:4:4:3:3)



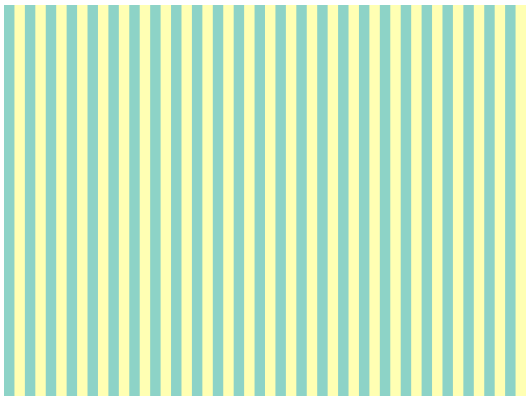
(e) GeForce GTX TITAN X (4:4:4:4:4:4)

Figure B.3.: Binning patterns used by the Nvidia Maxwell architecture.

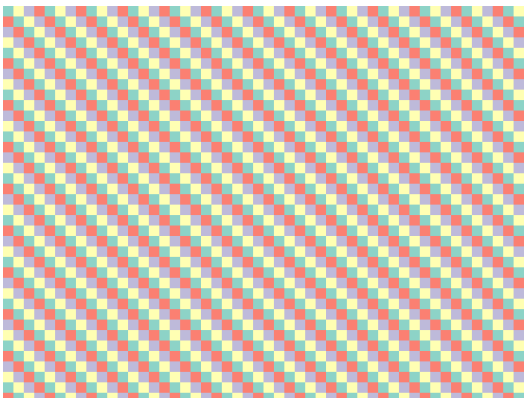
B.4. Nvidia Pascal



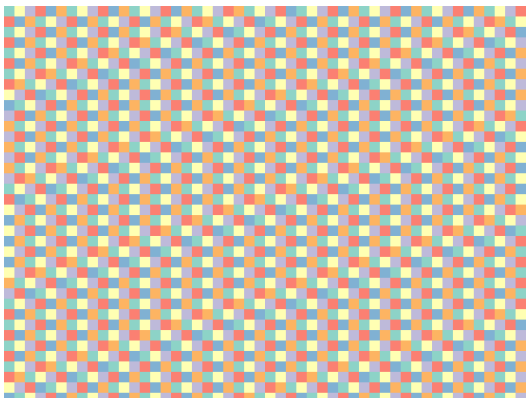
(a) GeForce GTX 1050 Ti (3:3)



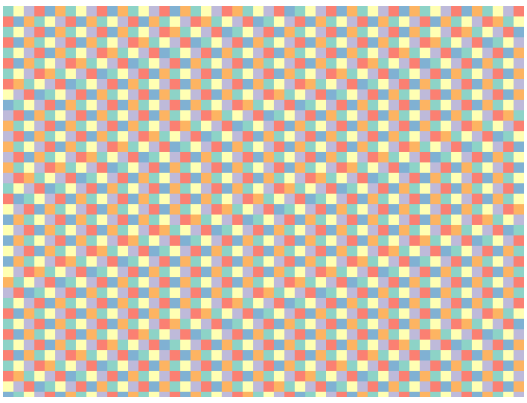
(b) GeForce GTX 1060 6GB (5:5)



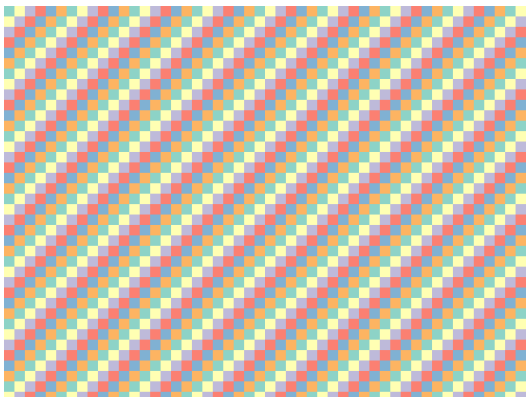
(c) GeForce GTX 1080 (5:5:5:5)



(d) GeForce GTX 1080 Ti (5:5:5:5:4:4)



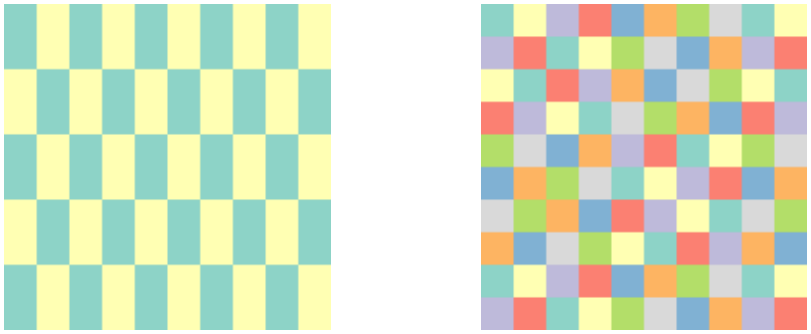
(e) TITAN X (5:5:5:5:4:4)



(f) TITAN Xp (5:5:5:5:5:5)

Figure B.4.: Binning patterns used by the Nvidia Pascal architecture.

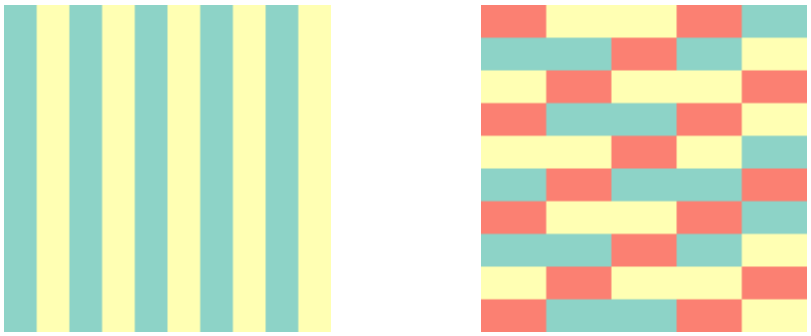
B.5. AMD



(a) Radeon 6770M (b) Radeon R9 270X

Figure B.5.: Binning patterns used by AMD GPUs.

B.6. Intel



(a) HD Graphics 4000 (b) HD Graphics 530

Figure B.6.: Binning patterns used by Intel GPUs.