Martin Winter

GPU-autonomous Dynamic Graph and Memory Management

DOCTORAL THESIS

to achieve the university degree of
Doktor der technischen Wissenschaften

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Markus Steinberger, BSc
Institute for Computer Graphics and Vision, Graz University of Technology

Referee

Prof. Dr. John Douglas Owens
Department of Electrical and Computer Engineering, University of California, Davis

Graz, Austria, July 21, 2021

To Vera, Gerlinde, Walter, Andreas and Anna

We are going to die, and that makes us the lucky ones. Most people are never going to die because they are never going to be born. The potential people who could have been here in my place but who will in fact never see the light of day outnumber the sand grains of Arabia. Certainly those unborn ghosts include greater poets than Keats, scientists greater than Newton. We know this because the set of possible people allowed by our DNA so massively exceeds the set of actual people. In the teeth of these stupefying odds it is you and I, in our ordinariness, that are here. We privileged few, who won the lottery of birth against all odds, how dare we whine at our inevitable return to that prior state from which the vast majority have never stirred?

Richard Dawkins, Unweaving the Rainbow: Science, Delusion and the Appetite for Wonder (2000)

Abstract

Dynamic memory allocation or dynamic graph management on a single instruction, multiple threads architecture, like the Graphics Processing Unit (GPU), is generally understood to be a challenging topic. On current GPUs, hundreds of thousands of threads might be concurrently active, allocating new memory, altering adjacency data, or freeing resources again. In both cases, sensible solutions must contend with thread contention, synchronization overhead, memory fragmentation and work balancing. Typical implementation guidelines even caution against the use of dynamic memory on the GPU, as data structure must rise to the challenge of thousands of concurrently active threads trying to allocate memory.

In an effort to utilize the parallel compute power of the GPU for more and more problem domains, one frequently encounters the need for dynamic memory in existing application domains. This is especially true when considering dynamic graphs, which are commonly used to analyse the increasing and ever-changing data arising across multiple fields. This includes communication networks (monitoring mobile phones and their connections to cell towers), social-media networks and web graphs (tracking friend relations or re-tweets *etc.*), intelligence networks (modelling agents and their communications), biological networks (e.g., folding proteins to gain insight into viruses like SARS-CoV-2) and many more. Since all these problems deal with huge networks, the GPU, as a data parallel processor, has potential to open up vast performance improvements.

In this thesis, we propose one base design as well as two evolutions to provide a GPU-autonomous solution to dynamic graph and memory management. *aimGraph* runs independent of the CPU after initialization, storing adjacency information on linked edge blocks, retaining memory locality within one block while still being able to handle the highly dynamic nature of dynamic graphs. *faimGraph* evolves this concept by significantly improving memory efficiency by introducing memory re-use queues and new and improved update strategies as well as work balancing measures usable by algorithms traversing the structure. Furthermore, it is the first framework to introduce the concept of dynamic

vertices additional to dynamic edges. Lastly, we broaden the concept of our memory re-use queues and GPU memory manager to general purpose memory management in our final iteration called *Ouroboros*. By virtualizing the memory re-use queues, we marry the benefits of memory efficiency and access performance into one final design. Due to its structural similarities with *faimGraph*, we also present a variant of *faimGraph* based on *Ouroboros*, called *ouroGraph*. This design significantly improves memory efficiency and algorithmic performance at a slightly higher update cost.

Our early design in *aimGraph* provides higher update rates and much improved initialization performance compared to related work at the time, while also ridding itself of the shackles of a tight CPU bound. *faimGraph* itself improves in all aspects on this design, showing best initialization performance, great edge update performance compared to *cuSTINGER*, *Hornet* and *GPMA* as well as superior algorithmic performance on PageRank and static triangle counting. *Ouroboros* is the most memory efficient memory manager on the GPU according to an in-depth survey of all memory managers on the GPU, with its page-based variants even providing best performance over a large range of possible allocation sizes. Its graph-adaption *ouroGraph* further improves upon the efficiency of *faimGraph* and increases the ease of porting algorithms as well as their performance.

Kurzfassung

Dynamische Speicherverwaltung sowie das Management von dynamischen Graphen auf der Graphikkarte (GPU), einer "single-instruction, multiple threads" Architektur, wird allgemein als herausforderndes Problem angesehen. Auf aktuellen GPUs sind potenziell hunderttausende Threads gleichzeitig aktiv, allokieren neuen Speicher, ändern Adjazenzdaten in einem Graphen oder geben Ressourcen wieder frei. In allen Fällen müssen sich effiziente Lösungen mit Thread-Konflikten, Synchronisationsaufwand, Speicherfragmentierung sowie gerechter Arbeitsaufteilung auseinandersetzen. Typische Implementierungsrichtlinien warnen sogar vor der Verwendung von dynamischem Speicher auf der GPU, da Datenstrukturen dem Allkokationsdruck von tausenden Threads gewachsen sein müssen, welche gleichzeitig Speicher allokieren und wieder freigeben.

In dem Bestreben, die parallele Rechenleistung der GPU für immer mehr Probleme zu nutzen, stößt man häufig auf den Bedarf an dynamischem Speicher in bestehenden Anwendungsdomänen. Das gilt insbesondere für dynamische Graphen, die üblicherweise zur Analyse der zunehmenden und sich ständig ändernden Daten verwendet werden, die in mehreren Gebieten auftreten. Dies umfasst Kommunikationsnetze (verwalten von Mobiltelefonen und deren Verbindungen zu Mobilfunkmasten), Social-Media-Netzwerke und Webgraphen (aufzeichnen von Freundschaftsbeziehungen oder Re-Tweets usw.), Nachrichtennetzwerke (das Modellieren von Agenten und deren Kommunikation), biologische Netzwerke (Faltung von Proteinen, um Einblicke in Viren wie SARS-CoV-2 zu erhalten) und vieles mehr. Da all diese Probleme mit großen Netzwerken zu tun haben, kann die GPU als daten-paralleler Prozessor enorme Leistungsverbesserungen ermöglichen.

In dieser Arbeit schlagen wir ein Basisdesign sowie zwei Weiterentwicklungen vor, um eine GPU-autonome Lösung für die dynamische Graphen- und Speicherverwaltung vorzustellen. *aimGraph* wird nach der Initialisierung unabhängig von der CPU ausgeführt. Dabei werden Adjazenzinformationen auf verknüpften Speicherblöcken gespeichert, wobei

die Speicherlokalität innerhalb eines Blocks beibehalten wird, während die hochdynamische Natur dynamischer Graphen weiterhin bestehen bleiben kann. *faimGraph* entwickelt dieses Konzept weiter, indem die Speichereffizienz erheblich verbessert wird. Es werden dabei Queues zur Wiederverwendung von Speicher und neue, verbesserte Aktualisierungsstrategien sowie Maßnahmen zur Arbeitsaufteilung eingeführt, die von Algorithmen verwendet werden können, welche die Struktur traversieren. Darüber hinaus ist es das erste Framework, welches das Konzept dynamischer Knoten zusätzlich zu dynamischen Kanten einführt. Zuletzt erweitern wir das Konzept unserer Queues für die Wiederverwendung von Speicher und des GPU-Speichermanagers auf die allgemeine Speicherverwaltung in unserer letzten Iteration mit dem Namen *Ouroboros*. Durch die Virtualisierung der Queues für die Wiederverwendung von Speicher vereinen wir die Vorteile der Speichereffizienz und der Zugriffsleistung in einem finalen Design. Aufgrund seiner strukturellen Ähnlichkeiten mit *faimGraph* präsentieren wir auch eine Variante von *faimGraph* basierend auf *Ouroboros*, genannt *ouroGraph*. Dieses Design verbessert die Speichereffizienz und die algorithmische Leistung bei etwas höheren Aktualisierungskosten erheblich.

Unser erstes Design, *aimGraph*, bietet höhere Aktualisierungsraten und eine deutlich verbesserte Initialisierungsleistung im Vergleich zu ähnlichen Projekten zu dieser Zeit und befreit sich gleichzeitig von den Fesseln einer engen CPU-Bindung. *faimGraph* selbst verbessert alle Aspekte dieses Designs und zeigt die beste Initialisierungsleistung, eine hervorragende Kanten-Update-Leistung im Vergleich zu *cuSTINGER*, *Hornet* und *GPMA* sowie eine überlegene algorithmische Leistung bei zwei Algorithmen. *Ouroboros* ist laut einer eingehenden Untersuchung aller Speichermanager auf der GPU der speichereffizienteste Speichermanager. Seine "page"-basierten Varianten bieten sogar die beste Leistung über einen großen Bereich möglicher Allokationsgrößen. Seine Graphen-Adaption *ouroGraph* verbessert die Effizienz von *faimGraph* weiter und simplifiziert die Portierung bestehender Algorithmen und erhöht deren Leistung.

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.

The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.

Date	Signature

Acknowledgments

After completing such a long project, one starts to reminisce about the last years and what people and events lead to this current moment. One important person, without whom the last seven years of my life certainly would not have been as joyful and fulfilling, is (my soon-to-be wife) Vera. You always supported me and made me laugh when I was too focused on a problem and found it hard to relax, I cannot imagine life without you, and I am the happiest person alive to be able share my love with you. I am also incredibly thankful to my parents, Gerlinde and Walter. Both supported me from the start, always encouraged me to strive for something greater, to use my talents and focus and who also allowed me to focus my energy on my studies, without the need to work for most of my university degrees. Both have been an inspiration to me for my whole life, the way they conduct themselves, how much of their energy and love they focused on us children, both are role models for me. Also, my brother, Andreas, and my sister, Anna, both are very special and dear to me and play an important part in my life, I absolutely love to make/enjoy music together or go for a hike or a bike ride, they simply are the best.

Switching from family to friends, I would like to first and foremost thank Markus Steinberger, who I not only consider as a mentor, but also a friend. Working with Markus was (and continues to be) a pleasure, his breadth of knowledge and amazing ideas but also his good-natured humor and caring for his students made the last few years really special to me. Sharing an office with Daniel also was one of the best parts of the last years and one of the things I missed most during the current home office period. I learned so much from Daniel and had a lot of fun, which let the hours pass by without much notice. The same is true for Jörg, who is always up for an interesting discussion, shares a lot of my interests and always helped me with all kinds of Linux-related questions. I also like to thank all other colleagues, which stopped by our office frequently for fruitful discussions and some light-hearted chat, which includes Mathias, Alexander and Philip. I am also incredibly thankful to Thomas, Daniel, and Thomas, with whom I have spent the vast

majority of my Bachelor and Master years at university, I learned an immense amount from all three and had lots of fun while doing so, I would not have been able to finish my studies that quickly without their support.

Lastly, I would also like to thank my external referee, Prof. John D. Owens, who spared no effort and helped me enormously to improve this thesis with thoughtful comments and lots of advice on potential improvements.

Contents

List of Figures

List of Tables

1 Introduction

Contents

1.1 Motivation

Dynamic graphs are commonly used to model and analyse the large, ever-changing data arising across multiple fields. This includes *communication networks*, where vertices model mobile devices with the connections between them or cell towers, represented by edges; *social-media networks*, where vertices may represent people with edges indicating friend relationships; and *intelligence networks*, where vertices model agents with edges highlighting their interactions. In combination with the rise of *big data*, there is an imminent need for highly efficient, dynamic graph data structures that support millions of vertices and edges, which can change constantly.

As the modern *Graphics Processing Unit* (GPU) becomes ever more ubiquitous and comparatively inexpensive, the GPU seems predestined to deal with this large-scale problem domain. Leaving aside the cost factor, the turn to massively parallel architectures like the GPU is also justified by limitations of hardware manufacturing: In the past, it was expected that every new hardware generation increases its clock speed and transistor count. Although the transistor count keeps growing exponentially, the clock speed has hit the so-called *power wall* [54]. This means that further increases in clock speed are bounded by the power and thermal limitations of the chips and lead to clock speed stagnation on modern *Central Processing Unit* (CPU) architectures. The GPU, on the other hand, has

still some thermal potential left and still sees minor growth in clock speeds in recent generations. Nonetheless, a significant speed-up is only achieved by exploiting parallelism. A trend towards rapid increases of core counts can especially be observed on the GPU, where new generations increase core counts by multiple thousand (e.g., considering the top range consumer offerings from NVIDIA, the 1080 Ti has 3584 cores, the 2080 Ti has 4352 cores and the newest 3080 doubles that to land at 8704 cores).

In general, the GPU has gained a lot of traction as a general purpose processor. Its *Single-Instruction Multiple Data* (SIMD) processing model lends itself to problems with high data parallelism, which is typical also in graphs, especially as they increase in size. As graphs are increasing in size, the available data parallelism increases as well. Furthermore, the throughput-oriented architecture of the GPU also fits the graph domain well. The GPU utilizes thousands of threads, drawing from comparatively small caches and relying on much simpler control logic compared to the CPU. Since most graph algorithms are comprised of simple operations that have to be performed on millions of objects, the GPU seems like an ideal candidate. Thus, it is not surprising that various *static graph libraries* target the GPU [14, 15, 31, 39, 58].

Turning towards dynamic memory management in general, one of the major hurdles in converting an existing, dynamic algorithm from the single-threaded CPU-side onto the GPU is concerned with handling of dynamic memory. In a single-threaded environment, the usage of dynamic memory provides a sensible way of dealing with a highly dynamic application domain. Solving this same problem in a highly concurrent setting, as on the GPU, is justifiably hard. Many modern memory managers on *CPUs* [9, 10, 19] build on one base design. They have one arena per CPU core to improve CPU cache hit rates, use a *mutex* for concurrent allocation as well as deallocation and manage memory in chunks.

A straightforward application of CPU algorithms, designed to deal with orders of magnitude fewer threads, often does not perform well. Basic solutions typically solve this problem by either over-allocating memory or performing expensive precomputations to estimate memory requirements. Such workarounds are not necessary for CPU computing, as dynamic memory allocation can be done efficiently, since the number of concurrently active threads rarely exceeds the double digit zone. However, on massively parallel, single instruction, multiple threads architectures, like the GPU, thousands of concurrently active threads may access and reallocate resources as well as change their allocation status by allocating new memory or releasing unused memory back to the system. Existing dynamic memory solutions are either limited in scope, e.g., they focus on single allocation sizes, or use data structures not well suited for concurrent manipulation. Implementation guidelines typically advise against the use of dynamic memory on the GPU. But not all applications lend themselves to precomputation of resource requirements or can be run with CPU interference to use the CPU for memory allocation. Given the allocation pressure on modern *GPUs* generated by thousands of active threads, allowing for efficient memory allocation is challenging. Additionally, keeping memory fragmentation low, as GPU memory still is a comparatively scarce resource, and avoiding costly CPU round-

trips adds to this challenge. The combination of the pressure on modern *GPUs* generated by thousands of active GPU threads with the requirement to keep memory fragmentation low and avoid CPU round-trips poses a significant challenge.

1.1.1 Research Questions

Inspired by these challenges, we formulate a number of research questions that we would like to answer in this dissertation.

1. Is it possible for a dynamic graph framework to run completely autonomously on the GPU? Given a massively-parallel architecture like the GPU and its high data parallelism suitable for large data sets, previous work still requires a tight bond to the CPU for memory management tasks. Can we design a solution for dynamic graph management that runs independently and autonomously on the GPU, managing its own memory to allow for efficient edge updates on different graph types as well as supporting algorithms running on such a structure?

2. Is it possible to design a dynamic graph framework for the GPU that truly allows for arbitrary growth and shrinking of both vertices and edges? Previous approaches treat vertices and edges separately and only allow dynamic changes to adjacency data, which prohibits the arbitrary use of the entire GPU memory for the complete graph. Can we keep the GPU-autonomy of the design presented in (1) while also allowing dynamic changes to both vertices and edges alike. Furthermore, can we improve memory efficiency by re-using both freed vertices and edge blocks and build a work balancing scheme usable by algorithms?

3. Is it possible to generalize dynamic memory management on the GPU, currently used only for specific user data, like vertices and edges, to general purpose memory management and its management structures? Previous work, especially when focussing on high performance, require significant portions of static memory for its management data, effectively reducing the amount of available memory to the system even if not needed. An ideal solution would allow for nearly the entire memory to be used by the application. Can we virtualize the management structures at hand to increase memory efficiency while still retaining the benefits of high allocation/deallocation performance and reduced memory fragmentation?

4. Is it possible to use this generalized approach for dynamic graph management as well to increase memory locality as well as algorithmic performance? A linked structure implicitly incurs overhead during traversal and limits the options to balance work efficiently during adjacency traversal as is common in many algorithms. Given this design, can we build a graph framework on the GPU that stores its adjacency on power-of-two aligned pages, keeps the dynamic nature of both vertices and edges while greatly increasing portability and algorithmic performance?

1.2 Dynamic Graph Management on GPUs

Managing a large number of entities—as required for *dynamic graphs*—is challenging on the GPU. The GPU performs best when memory requirements and layout are known beforehand to allow appropriate optimizations. Especially, changing memory requirements are difficult to handle. Typically, memory allocation on the GPU is handled by the CPU, which disrupts parallel execution on the GPU to give way to sequential allocation on the CPU. While efficient memory allocation already forms an issue for dynamic GPU execution, freeing memory is an even bigger challenge: As many small memory deallocations can yield large overheads, freed memory is often simply not returned to the system. Over time, such strategies lead to memory fragmentation, reduce the available memory, and ultimately lead to system failure as it runs out of memory.

Furthermore, unbalanced graphs lead to unbalanced work loads. As the structure of dynamic graphs continuously changes, load balancing strategies also need to adapt to achieve high performance. Thus, performing load balancing with performance influencing factors in mind, like thread divergence on the *SIMD* cores of the GPU and memory locality, becomes an even more challenging task for dynamic graphs. Probably due to these issues, the number of dynamic graph frameworks for the GPU is limited (not including our own work), namely, *cuSTINGER* [22], *GPMA* [46], *Hornet* [13] as well as *hashGraph* [3]. Only *hashGraph* supports fully dynamic graphs; the others consider the graph's vertices fixed and only support dynamic edge data and/or updating individual properties of existing vertices. Each solution comes with its own drawbacks, both *cuSTINGER* and *Hornet* heavily rely on CPU-synchronization for their memory allocation routines, while both *GPMA* and *hashGraph* store its adjacencies not in contiguous memory, potentially decreasing algorithmic performance. Especially *cuSTINGER* and *GPMA* furthermore can have excessive memory requirements due to lavish use of memory, leading to worsening memory management strategies over time or even system failure due to out-of-memory.

1.2.1 Contributions

We present two versions of our dynamic graph management system running autonomously and independently on the GPU.

Our first foray into dynamic graph management is called *aimGraph*. *aimGraph* runs after an initial allocation from the CPU autonomously on the GPU, storing adjacency information on fixed-size edge blocks. These edge blocks can be linked together to allow for larger adjacencies. This design still provides great memory locality on an edge block while still retaining the dynamic requirements posed by the problem domain. All further details of this design are discussed in length in Chapter 3.

While this already provides a suitable solution for dynamic graph management on *GPUs*, it still leaves some open challenges, as memory management cannot re-use freed up edge blocks and vertices remain static within this design. We tackle these issues in the first

evolution of this concept called *faimGraph*, presenting a *fully dynamic* GPU framework for large dynamic graphs, which is completely GPU-autonomous, reclaims and reuses all freed memory, and reduces fragmentation to a minimum for both vertex and edge data. To achieve these goals, we introduce an advanced dynamic memory management system for the GPU tailored to graphs, which uses efficient *queueing* structures to reassign memory directly on the GPU in *O(1)*. Although more complex data management naturally increases memory access times, our framework achieves equal or better performance throughout both memory management routines and algorithms executing on top of the graph structure. All intricacies are detailed in Chapter 4.

Lastly, building on our efforts developed for dynamic memory management on the GPU, called *Ouroboros*, we present this final design, combining the benefits of contiguous memory per adjacency and GPU-autonomy, marrying the concepts found in *faimGraph* and *Ouroboros* into one, called *ouroGraph*. As *Ouroboros* already leaves the region right after its memory manager as linearly addressable memory, we can directly integrate all dynamic graph management functionality as introduced in *faimGraph*. Adjacency data is now stored on power-of-two aligned pages, allocated from larger chunks through the memory manager. This not only drastically increases memory efficiency, especially for sparse graphs, but it also greatly increases algorithmic performance and in general memory access performance. This culmination of our work is discussed in detail in Chapter 6.

1.3 Dynamic Memory Management on GPUs

Approaches to handle dynamic memory management on *GPUs* first showed up around ten years ago with the introduction of dynamic memory management via the NVIDIA Toolkit [40]. Since then, various techniques and refinements have been proposed to speed up the allocation of memory, which has long been considered a challenging issue on a GPU. Deciding which approach fits which application best can be a challenge. Furthermore, new architecture features have been introduced (e.g., independent thread scheduling on the NVIDIA Volta [41] architecture). These enable new programming paradigms and shift the focus more towards thread-based computation, easing the conversion of an existing CPU algorithm to a GPU algorithm. These new capabilities might entail a rethinking of dynamic resource management in an environment with a stronger focus on individual thread allocation performance compared to allocations at a warp level.

One key problem in dynamic memory management is the question of how to manage a large number of resources. The two prevalent methods to keep track of large numbers of dynamic objects are *linked-lists* and *arrays*.

Linked-list-based approaches require each of the dynamic objects to offer at least a *next pointer* for traversal of objects. As any number of objects can be linked, linked-lists are only limited by the available memory. Their advantage is that the number of objects held by the list is only limited by the available memory. However, especially on the GPU, linked-lists come with many disadvantages, outweighing the advantages significantly. First,

update operations on a linked-list are inherently sequential, hence modifications with thousands of threads come with a significant slow-down. Second, the mandatory *next pointer* incurs a large overhead when managing small objects. Third, accessing linked-lists causes scattered memory accesses, deteriorating cache usage on the GPU. Recent work, as introduced by Ashkiani et al. [2], addresses this problem by introducing elaborate warp-cooperative work sharing strategies to reduce branch divergence during list traversal.

Array-based methods, on the other hand, are—due to their static size—inherently limited by the number of objects they can manage at a time, as this is determined at initialization. These can be efficiently implemented as queues on top of a ring buffer. Using $front$ and *back* pointers as well as atomics, efficient access to individual items can be organized. They provide efficient access to individual items and entail no per-object memory overhead. Furthermore, only currently re-usable objects store an identifier in the array. Compared to *linked-list-based* methods, *array-based* methods offer vastly greater parallel access capabilities and potentially increased performance. Many applications require support for *dynamic work generation*, as the amount of objects produced might depend on the actual content of the input. In such applications the number of simultaneously existing dynamic object often has a large variance or is even unpredictable, which requires significant over allocation of the arrays. Even if the maximum required array size can be computed for a specific input, the allocated memory might only be utilized for a fraction of the applications runtime. This means that a potentially large part of the allocated memory, is underutilized for most of the execution time.

1.3.1 Contributions

To address the aforementioned problems, we introduce *Ouroboros*: a new, dynamic memory management system for *GPUs*, based on novel, virtualized queues. We start with an array-based queue for memory reuse of a single, configurable page size. We extend this concept to support multiple, different page sizes by introducing chunks, a larger unit of memory. Chunks are broken up into pages, managed by one array-based queue per page size. To reduce the memory overhead, associated with multiple support data structures, we virtualize the queues, retaining their performance benefits over linked-list based methods.

We make the following contributions:

- To allow for differently sized allocations, we extend a simple, array-based memory manager [62], increasing the system's versatility. Splitting memory into equally-sized chunks that themselves can be subdivided into all desired page sizes, we store re-usable pages and chunks in queues. Re-usable pages or chunks are referenced in such queues for re-use. Bulk allocations of pages are handled using an optimized synchronization primitive [20].

- We propose two virtualized queues, storing the array-based queues themselves on memory chunks. Both can either manage page indices directly for maximum al-

location performance or use chunk indices (of chunks with free pages) and reduce memory requirements further:

  - Our *array-hierarchy, virtualized queue* retains a small chunk pointer array, as a hierarchy level over the actual queue, significantly reducing the memory overhead while retaining most of the performance benefits.
  - Our *linked-chunk, virtualized queue* removes the base data structure all together in favor of pointers to the beginning and end of the virtual queue, accepting a slight performance penalty in favor of drastically reduced memory overhead

- We integrate *Ouroboros* into *faimGraph* [62] to create *ouroGraph*, demonstrating the applicability and benefits of our allocator in a real application scenario resulting in improved initialization and algorithmic performance at comparable update performance, while retaining the possibility to have a second dynamic region in the same space, as done with dynamic vertices in *faimGraph*.

All details concerning our effort in the area of dynamic memory management can be found in Chapter 5.

As part of a larger survey on dynamic memory management on *GPUs*, we also evaluate a larger set of benchmarks. This goes back as early as 2010 with *XMalloc* [25], continues with *ScatterAlloc* [53] in 2012, *FDGMalloc* [59] in 2013, an approach by Vinkler and Havran [56], *KMA* [49] and *Halloc* [1] in 2014 as well as *BulkAllocator* [20] and *DynaSOAr* [50]. For each of these, we provide a short introduction and in the end, we thoroughly evaluate all, which are publicly available, on a large test suite. This includes test on allocation performance (both *thread-based* and *warp-based*), performance scaling, mixed allocation, fragmentation and out-of-memory performance as well as real world testcases, including a synthetic test and initializing dynamic graphs. Based on these data, we assess the feasibility of each approach and highlight the intricacies detected. In the end, we provide recommendations for the best usage scenarios. Overall, *Ouroboros* shows best memory utilization and the least memory fragmentation of all approaches. Evaluating allocation performance, *chunk-based* variants of *Ouroboros* perform middle of the pack due to their two-stage allocation design, with *page-based Ouroboros* performing best over a large range of allocation sizes.

1.4 Outline

The structure of this thesis follows the evolution of the base design as introduced with *aimGraph*. But first, related work is discussed in detail in Chapter 2. We focus on efforts on static graph frameworks on *GPUs* and then discuss current efforts in the direction of dynamic graph management on *GPUs* in detail. Furthermore, we present the current state-of-the-art in dynamic memory management on the GPU, followed by a short detour into *queue designs* on the GPU as well as some graph algorithms designed for the GPU.

The next section in Chapter 3 introduces our first design called *aimGraph*, where we highlight the memory layout used as well as update strategies. We evaluate our framework against *cuSTINGER*, testing memory initialization/footprint as well as edge update rates.

Continuing on with the first evolution of *aimGraph*, we present *faimGraph* in Chapter 4. We detail the basic memory layout, our queue design for memory re-use as well as our strategy for incorporating dynamic vertices. This is followed by an evaluation against *cuSTINGER*, *GPMA* and *Hornet*, evaluating memory initialization and footprint, vertex as well as edge updates and ends on performance measures on two graph algorithms.

Next, we broaden the ideas previously introduced to serve dynamic memory to general applications, which can be found in Chapter 5. Building on our queue design introduced in *faimGraph*, we build a dynamic memory manager on the GPU, called *Ouroboros*. We discuss two initial variants of this design and then, with a focus on memory efficiency, we introduce two more variants, which virtualize the base queue for even more efficient management. The evaluation is done on all publicly available, general purpose memory managers on the GPU, including *XMalloc*, *ScatterAlloc*, *Halloc*, *RegEff* and the native *CUDA-Allocator*. It includes testing thread-/warp-based allocation, mixed allocation, performance scaling, memory fragmentation as well as some real-world examples.

In the penultimate section, which can be found in Chapter 6, we show how one can utilize *Ouroboros* in the context of dynamic graph management and present *ouroGraph*. Adjacency data is stored on power-of-two aligned pages allocated from chunks, growing from the top down, while dynamic vertices still grow from the bottom up. In the evaluation, we show comparable update performance compared to *faimGraph* as well as superior algorithmic performance, as better memory locality and work balancing can be employed on contiguous memory as found in *ouroGraph*.

The last section found in Chapter 7 gathers all the threads together, summarizing the benefits and drawbacks of each approach and ends on an outlook on the future, detailing remaining challenges.

2 Related Work

Contents

Related work on graph data structures for the GPU can be categorized into static graph libraries (Section 2.1), dynamic graph libraries (Section 2.2), dynamic memory management on GPUS (Section 2.3), and GPU adapted implementations of graph algorithms (Section 2.4).

2.1 Static Graph Frameworks on GPUs

There exists a variety of static graph data structures on the GPU: *nvGraph* [39] (NVIDIA Graph Analytics library) offers implementations of three widely-used algorithms, supporting up to two billion edges (using an NVIDIA Tesla M40 with 24 GB). *BlazeGraph* [31] offers a high-performance graph database, using its own domain-specific language, *DASL*, to implement advanced analytics. *BelRed* [15] addresses the problem that manual effort is often required to build graph application on the GPU. They introduce a library of software building blocks which can be combined to build graph applications. *Gunrock* [58] is a *CUDA* library for graph processing using highly optimized operators for graph traversal while achieving a balance between performance and applicability. *GasCL* [14], a vertex-centric graph model for the GPU, written in *OpenCL*, supports the "think-like-a-vertex" programming model. *SEP-Graph* [57] presents a novel framework for graph processing on the GPU, which builds on a hybrid model that can switch between three pairs of parameters. These include (1) synchronous or asynchronous execution mode, (2) Push or

Pull communication, and (3) Data-driven or Topology-driven traversal of the graph. Each of these parameters is chosen to achieve the shortest exeuction time. *SIMD-X* [30] addresses the issue inherent in many algorithms like breadth-first search, which suffer from underutilization on GPUs by introducing a new programming model, called the *Active-Compute-Combine* (ACC) model, which filters out inactive vertices at runtime and can remap tasks to CUDA cores for enhanced workload balancing. Brahmakshatriya et al. [11] introduce an extension to the *GraphIt* compiler framework, called *G2*, that builds on the same specification but can be deployed on both CPUs and GPUs. Developers can write an algorithm once in this language and can then combine load balancing and other optimizations on the GPU without resorting to writing low-level code. *Tigr* [38] is a graph transformation framework that aims to transform existing, irregular (in the sense of power-law distributed) graphs into more regular graphs that fit the parallel execution model of the GPU better. *CuSha* [27] presents two new graph representations that try to overcome the irregular memory accesses and GPU underutilization common with CSR data layouts, called *G-Shards* (autonomous sets of ordered edges) and *Concatenated Windows* (building on these *shards* for even better utilization). While all these libraries achieve high performance for static graph algorithms, and ideas like building blocks and alternative programming models are also interesting for dynamic graphs, they do not consider dynamic changes of graphs. Thus, directly using any of these frameworks for dynamic graphs would require a complete reallocation of the graph whenever any part of the graph changes. This is obviously not sustainable as graphs are changing often and therefore specialized solutions for dynamic graphs are required.

2.2 Dynamic Graph Frameworks on the GPU

Due to the number of challenges of dynamic resource management on GPUs, the list of available frameworks is understandably comparatively smaller. But starting in 2016 with the introduction of *cuSTINGER* [22] by Green and Bader, a handful of frameworks tailored specifically for the GPU have been introduced, not including the solutions proposed in this thesis. *DCSR* [29] as introduced by King et al., which is designed for insertion-only updates and adapts the *CSR* format with linked edge blocks, is not discussed in more detail, as it does not support edge deletions or efficient queries. The order of the introduction follows the publication date of the individual approach.

2.2.1 *cuSTINGER*

The first approach specifically designed for dynamic graph management on GPUs is *cuSTINGER* [22], introduced in 2016 at *HPEC'16*. It is a GPU-adaption of *STINGER* [6] and its internal memory manager [18]. One adaption made during the transition to the GPU is a switch in the adjacency storage format from storing edges in an *AOS* approach to *SOA*. Especially for weighted or semantic graphs, this can improve memory access

performance on the GPU significantly. Furthermore, the list of edge blocks per node is replaced by a single adjacency array, once again designed to improve performance with SIMD memory access. Memory is still managed by the CPU by a memory manager, which groups together multiple adjacency lists into larger blocks to reduce the number of calls to `cudaMalloc()`. Additionally, different memory allocation modes can be chosen that trade off between storage utilization and a reduction in reallocation calls. In general, *cuSTINGER* supports three different graph types including simple, weighted and semantic graphs. Edge insertions try in an initial kernel to insert all updates into the corresponding adjacencies. Adjacencies that require augmentation are marked on the GPU and then reallocated on the CPU so that a second kernel launch can finish the insertion procedure. Edge deletions are simpler, as *cuSTINGER* does not decrease the size of adjacencies, but only shuffles edges to the front onto deleted positions.

Although *cuSTINGER* targets the GPU, its core memory management tasks still heavily require the CPU and many calls to `cudaMalloc()`. This happens during initialization as well as during edge insertion procedures. To reduce the probability of these costly CPU operations, *cuSTINGER* over-allocates all adjacency structures and as mentioned before, does not free unused adjacency space. This leads to higher memory fragmentation and memory waster overall and even potential system failure due to out-of-memory. When using conservative memory bounds, updating the graph involves significant overhead managed sequentially on the CPU. Additionally, only edge updates are supported within the system and vertices remain static.

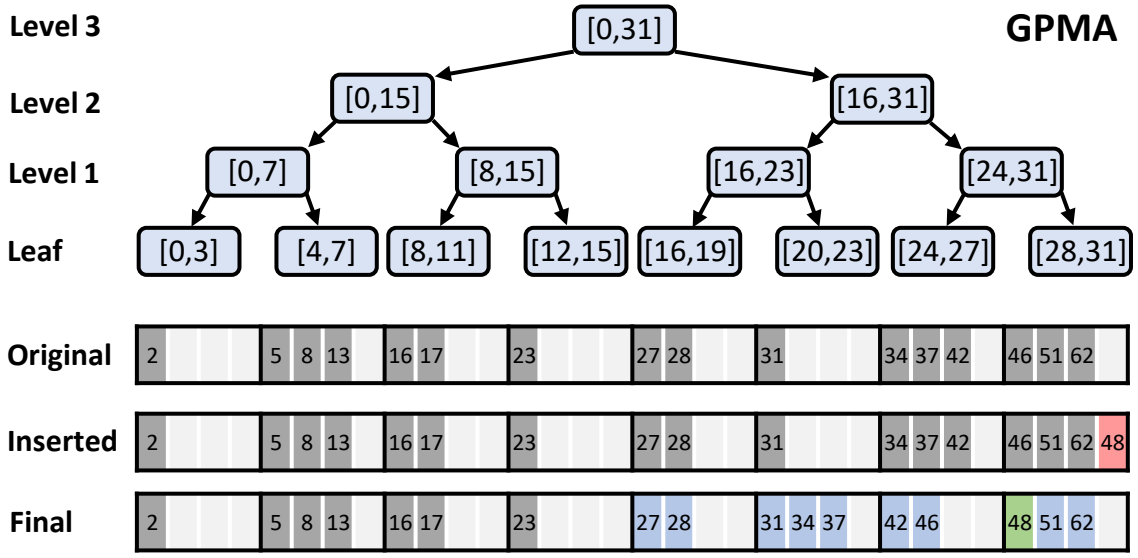
Nonetheless, *cuSTINGER* showed that it is possibly to utilize the massively-parallel capabilities of the GPU for dynamic graph management and also showed comparable performance for static triangle counting compared to an algorithm running on a *CSR* data structure.

2.2.2 *GPMA*

GPMA [46] was introduced by Sha et al. in 2018 and proposes a novel, dynamic data structure for graph management on *GPUs*. On a top-level, *GPMA* supports multiple streams operating on its data structure to either update edges or perform graph analytics tasks. The underlying data structure used is derived from a *Packed Memory Array*[8], as can be seen in Figure 2.1.

A *PMA* is a self-balancing tree structure which separates its memory space into leaf segments with $O(\log N)$ length. For each level in the tree, the *PMA* is designed to keep an update complexity of $O(\log^2 N)$ by choosing appropriate lower and upper density bounds. Once a segments falls out of the range of these density bounds, then the *PMA* tries to compensate by reallocating the elements in the segment's parent. This operation is propagated to the top until all segments once again fall into their respective density bounds.

This can be seen in Figure 2.1, where the insertion of node 48 causes the density of the last segment to breach its upper density bound. As the neighboring segment is also



	Leaf	Level 1	Level 2	Level 3
segment size	4	8	16	32
density lower bound ρ	0.08	0.19	0.29	0.40
density upper bound τ	0.92	0.88	0.84	0.80
min # entries	1	2	4	8
max # entries	3	6	12	24

Figure 2.1: *GPMA* maintains its graph in a so-called *Packed Memory Array* (PMA), which stores its entries in partially contiguous memory. Gaps are left to accommodate fast updates within the structure. *PMA* is a self-balancing binary tree structure that keeps its elements implicitly sorted. For each level in the tree, a lower and upper bound are chosen to achieve $O(\log^2 N)$ update complexity.

already at max capacity, this is propagated two levels up, which results in the final state at the bottom of the plot, where the range 16–32 is reallocated. Since this data structure is inherently sequential due to its rebalancing efforts, Sha et al. introduce two variants specific for the GPU. *GPMA* uses a thread-per-update design and a lock-based approach to guarantee consistency. All affected leaf segments are identified in advance and the updating threads compete for the locks, so that each updating operation is completed in a mutually exclusive fashion. Threads failing to acquire a mutex simply retry in the next iteration. Although this approach certainly works on a parallel processor like the GPU, it

has a number of drawbacks. Threads traversing the tree structure individually experience uncoalesced memory accesses; atomic pressure on the locks and the accompanying thread conflicts lead to retries in subsequent rounds.

To combat these issues, they introduce *GPMA*$^+$, a lock-free approach. The major changes to the algorithm include

- The updates are first sorted and their corresponding leaf segments are located
- Updates targeting the same leaf segment are grouped for processing level-by-level
- GPU primitives like warps or blocks or even the whole device are used to compute the necessary requirements depending on the segment size

This design completely forgoes locks, can achieve much better work balancing and better memory access patterns due to the grouping of updates focused on one memory region. Using a trick by inserting guard entries at adjacency boundaries into the *PMA*, the internal data structure can even mimic the common *CSR* storage format.

Overall, *GPMA* and *GPMA*$^+$ present two very interesting approaches to dynamic graph management on *GPUs*. The major benefit is the bounded update complexity and inherently maintained sort order. The main drawbacks also stem from the choice of this particular storage format. Concerning updates, depending on the update pressure on individual adjacencies, this can lead to frequent rebalancing efforts which move quite a lot of memory. Furthermore, as the whole graph is managed via a single *PMA*, if the storage requirements exceed the capabilities of the current *PMA*, the whole structure has to be re-build in a twice as large *PMA*, incurring significant overhead. Additionally, the framework does not natively support changes to vertices, neither vertex insertion nor deletion is available. Lastly, algorithms operating on the graph either have to pre-process the adjacencies to get contiguous memory as is demonstrated in the paper or deal with unbalanced work loads due to the holes left in the storage format.

2.2.3 *Hornet*

Improving upon the initial design of *cuSTINGER*, *Hornet* [13] by Busato et al. was introduced in 2018 at *HPEC'18*. Figure 2.2 shows an overview of the design for a small, weighted graph. The main management structures are still held and maintained by the CPU, but the overall design has been vastly improved, especially concerning memory efficiency.

Hornet use three building blocks to achieve this design:

- **Block Arrays:** these store multiple adjacency arrays
- **Vectorized Bit Tree:** keeps track of the allocation state within a *block array* to efficiently find and reclaim empty blocks

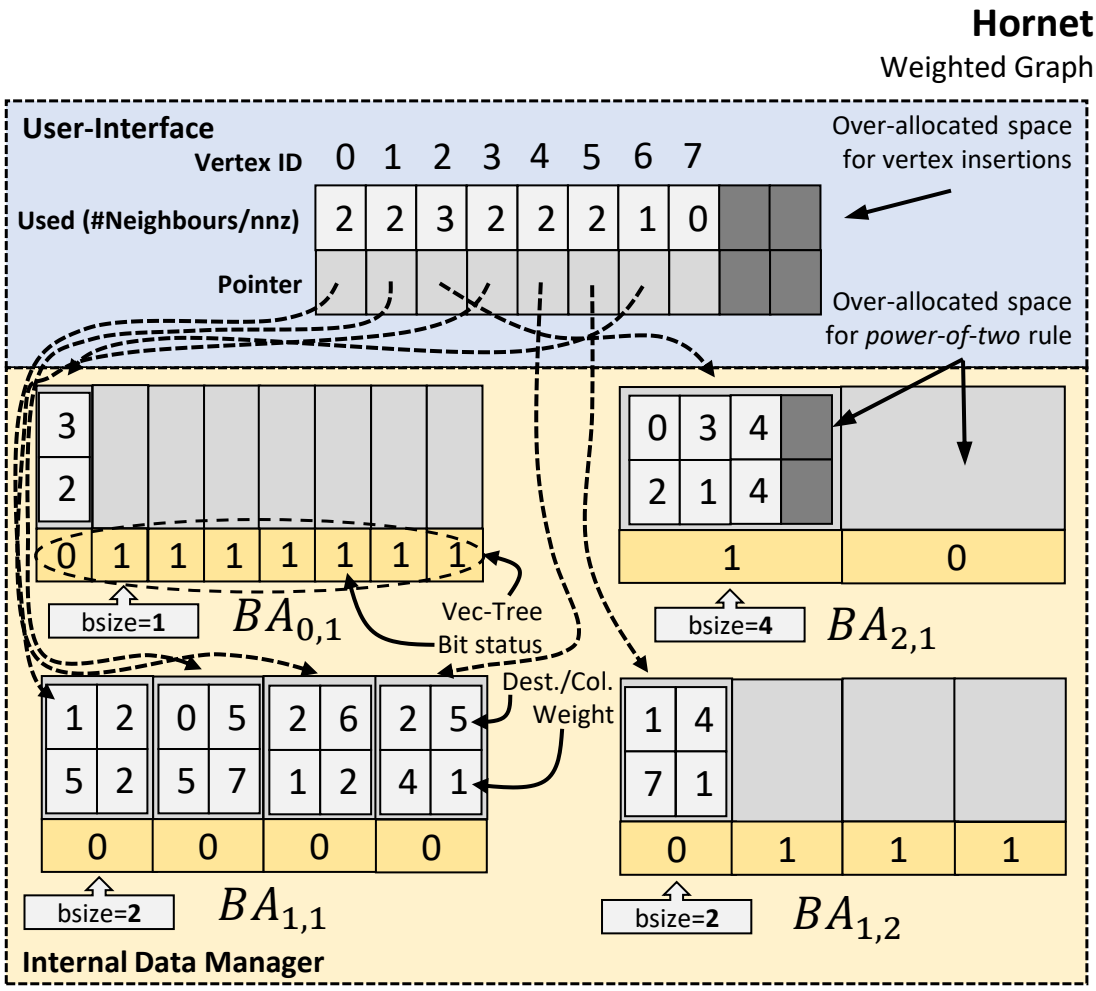


Figure 2.2: *Hornet* uses two-tiered storage system to store adjacencies. Large *block arrays* are allocated that can be split into power-of-two adjacency arrays to store the adjacency per vertex. Each *block array* tracks its allocation state via a *vectorized bit tree* and all *block arrays* per adjacency block size are tracked furthermore in a B^+ tree.

- **B$^+$ Tree**: One tree for each size supported by the system, managing the individual *block arrays*

Block arrays are equally-sized chunks of memory (similar to chunks or *Superblocks* later discussed in Section 2.3) and contain the individual adjacency arrays, each of them equal to a power of two in size. All adjacency arrays on one *block array* have the same size (a power of two); this bounds the memory requirements for a graph as $2 \cdot |E|$. As *block arrays* might only be partially filled or adjacency arrays are deallocated, a *vectorized bit tree* is used to keep track of the allocation state of the individual blocks on each block array. It ensures that *block arrays* are fully utilized before new allocations are necessary, reducing

the overall memory requirements and allowing for efficient finding of empty blocks. These trees are implemented as vectors of boolean values storing the *logical OR* of its children, hence simply inspecting the root already contains the information if a specific *block array* has empty blocks left. The *block arrays* per size are overall managed by a B^+ tree to ensure scalability and efficiency. This results in an array of B^+ trees for the whole graph.

Initialization consists of three phases, where first the graph is built on the CPU by finding corresponding *block arrays* and temporarily storing the adjacency data in these already on the CPU. In the last step, the whole *block arrays* are copied to the GPU for better memory bandwidth utilization. Updates are handled in batches, which are first pre-processed, which includes sorting and *cross duplicate removal*, then all vertices in need of reallocation get new memory and the current state is copied over and lastly, the new edges can be inserted or deleted from the graph.

Hornet greatly improves upon *cuSTINGER* especially considering initialization time and memory efficiency. Also, especially for larger update batch sizes, performance also greatly increases as *Hornet* is able to pre-process all adjacency changes much more efficiently before going for a round-trip to the GPU. Nonetheless, *Hornet* still requires a tight bond with the CPU, as all management tasks are still initiated and controlled on the host. This means that initialization still is quite slow, as most work has to be done by the CPU. Like *cuSTINGER*, vertex data is considered static and no update strategies considering dynamic vertices are present within the framework. Furthermore, especially for sparse graphs, significant changes to the adjacency still form major overhead as this requires significant reallocation on the CPU, similar to the initialization stage. But considering algorithms, performance even improves compared to *CSR*, likely due to better locality of vertices with similar degree.

2.2.4 *Dynamic Graphs on the GPU*

The newest addition in the line of dynamic graph management, not considering our own work *Ouroboros*, is called *"Dynamic Graphs on the GPU"* [3] by Awad et al., henceforth referred to as *hashGraph* in short.

They introduce a dynamic graph management system on *GPUs* based on high-performance hash tables for adjacency storage, which sets itself apart from the other approaches (except our own work, *faimGraph* and *ouroGraph*) as it not only supports edge updates, but also vertex updates. Furthermore, they also establish an evaluation strategy designed to better highlight real-world use of dynamic graphs. Lastly, they also integrate their dynamic graph data structure into *Gunrock* [58].

The identify the storage of the per-vertex adjacencies as the most important factor in designing a dynamic graph framework. Each vertex manages its own adjacency list, which could be built as a hash table [2] or even a B-Tree [4]. Due to the focus of this work on high update and query throughput, hash tables are chosen as the underlying data structure. The hash table implementation is based on *Slab Hash* [2] and exists in two

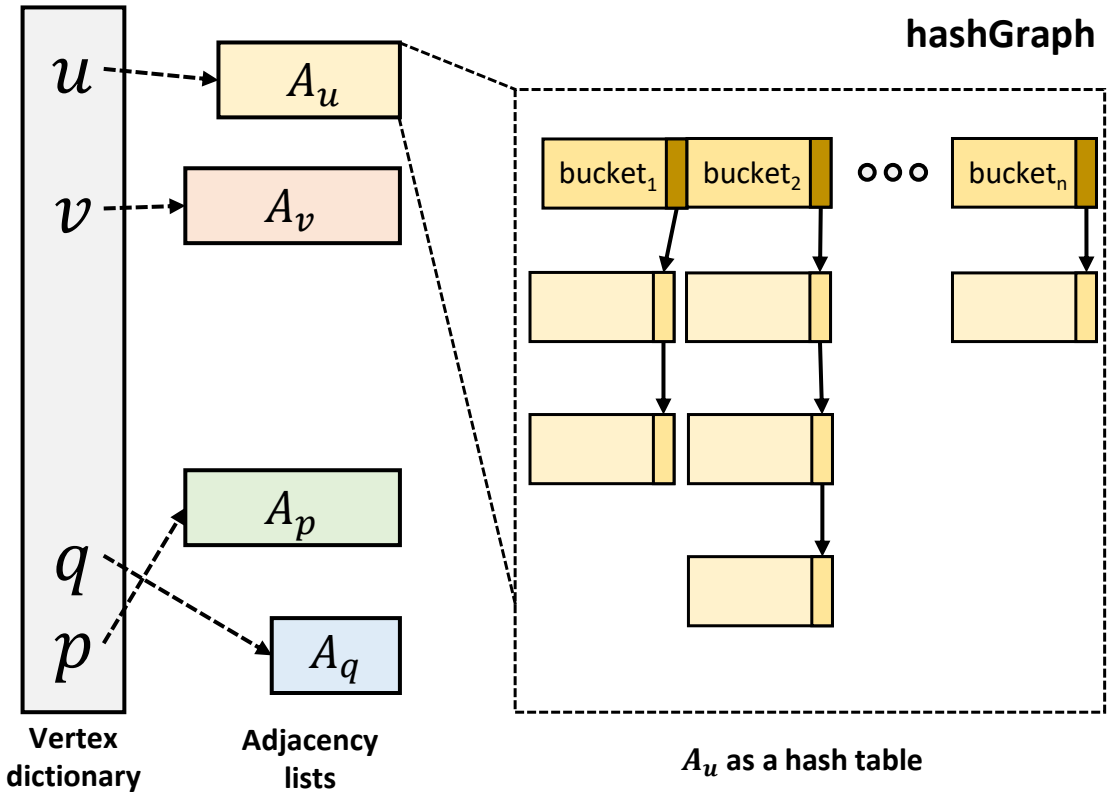


Figure 2.3: Overview of *hashGraph*, which uses hash tables to store adjacency data. New buckets for the hash tables can be allocated using a dynamic memory manager.

variants to support simple and weighted graphs. Vertices are stored in a fixed-size array, indexed by the vertex index, which can be increased if needed (but frequent reallocation is minimized by over-allocating this region). Adjacency lists are stored in one hash-table per vertex, given a static load factor (0.7 as chosen in the paper) and a bucket capacity (15 or 30 depending on the graph type) and the number of edges per adjacency, the number of buckets per hash table can be computed. Choosing the right load factor is significant for performance but also memory overhead. During an edge insertion procedure, if a hash table becomes full, each hash table can also dynamically allocate additional slabs using a dynamic memory manager.

In the initialization stage, the initial requirements are not allocated individually to reduce the overhead, but are grouped together and allocated in one bulk allocation. During operation, all operations implement the *Warp Cooperative Work Sharing (WCWS)* strategy, where each thread has an independent task assigned but all threads in a warp cooperatively solve one task after the other. This execution pattern fits the coalesced memory access requirements of the GPU quite well and results in better update performance. Each vertex can provide an adjacency list iterator, which can be used to query if edges exist (loop over all buckets associated with this vertex, loading one slab at a time

and moving from slab to slab using the `next` operator). During edge update operations, threads on a warp-level cooperate during the update procedure by first determining if multiple edges share the same source, allowing for grouping of these operations. Edge deletion only marks deleted edges with a *tombstone* so as to guarantee uniqueness within the slab hash. These can later be flushed completely from the data structure and edge insertions ignore these locations for better performance. This results in memory overhead at the cost of higher throughput. Vertex updates are also supported, which includes adding a new vertex and a set of edges connected to it but also deleting all reference to a vertex within a graph during vertex deletion.

During their evaluation, they focus on updates for both edges and vertices, showing great performance gains over *Hornet* and *faimGraph*, as expected from a hash table. They also introduce two new workloads for evaluation, a so-called *bulk build* (inserting all edges in one batch) and *incremental build* (starting with an empty graph and inserting edges incrementally), once again showing great build performance compared to *Hornet*. Lastly, they consider static and dynamic versions of *triangle counting*, where performance for very large graphs can be similar to *Hornet* or *faimGraph* but typically lacks behind as not sort order can be enforced.

Overall, *hashGraph* is a exciting new approach to dynamic graph management with a particular focus on edge update and query performance. It is also able to handle bulk and incremental builds very well. Potential drawbacks include the memory overhead associated with the load factor as well as the decision not to re-use deleted edge positions. Furthermore, the reliance on a dynamic memory manager for slab allocation and a less than optimal memory layout for typical vertex-based algorithms might also pose some problems in certain scenarios. Nonetheless, *hashGraph* especially suits applications in need of fast insertion, deletion as well as look-ups.

2.3 Dynamic Memory Management on GPUs

The following sections discusses currently presented memory managers on the GPU as well as a commonly used data structure, see Section 2.3.10, to manage dynamic resources. All of the memory managers offer some variant of the standard *malloc/free* interface and operate on a large block of memory with a configurable size. Each approach is given a shorthand for better readability, which will be used throughout the thesis. Any performance evaluation is deferred to Section 5.6.

2.3.1 CUDA Allocator

NVIDIA initially introduced its allocator [40] (henceforth referred to as *CUDA-Allocator*) as early as 2010 for GPUs of compute capability 2.0. It implements the standard *malloc/free* interface and is accessed on a per-thread level. Newer additions include *__nv_aligned_device_malloc*, which allocates aligned memory, aligned to a non-zero power

Name	Build	General Purpose	Re-use Primitive	Design Goal
CUDA-Allocator	all	✓	?	Reliable Memory Allocation
XMalloc	< `7.0`	✓	Lists FIFO-queue	Coalesce Requests Small Re-use Buffers
ScatterAlloc	< `7.0`	✓	Lists Bitmaps	Scatter Memory Access using Hashing
FDGMalloc	< `7.0`	∼	Lists	Warp-based Allocations for Temporary Memory
RegEff	< `7.0`	✓	Circular List Flags	Register-Efficient Lightweight Allocations
Halloc	< `7.0`	✓	Head Pointer Bitmaps	Warp-aggregated Atomics using Hashing
KMA	*OpenCL*	✓	Lock-Free Queue Bitmap	Two-layer Memory Manager for *OpenCL*
DynaSOAr	all	✗	Hierarchical Bitmap	Allocate pre-defined structures in SOA
BulkAllocator	≥ `7.0`	✓	Per-Size Bin List Static Binary Tree	Two Allocators based on Bulksemaphore
Ouroboros	all	✓	Per-Size (Virtualized) Queues	(Virtualized) Queues for Efficient Memory Re-use

Table 2.1: Condensed overview of all published memory managers for the GPU.

of two. There is unfortunately very little information available on the implementation, which only allows for speculation as to its internal structure. Its major benefit is the usability regardless of allocation size required and its thread-based allocation model. It does not natively support any group-based allocation procedures and can also only be initialized once with a given size (increasing this memory requires destroying the current context). Performance is generally considered weak, but reliability is key.

2.3.2 XMalloc

XMalloc [25] is the first, non-proprietary, dynamic memory allocator for GPUs, introduced also in 2010. Its main contribution is the coalescing of allocation requests on the SIMD width for faster, lock-free FIFO queues.

Large allocations (as well as *Superblocks*) are served from a heap, which is segmented into free and allocated *Memoryblocks*, as can be seen in Figure 2.4. These blocks form a linked-list, which allows for merging of neighboring blocks. This type of allocation is relatively slow, as the list of memory blocks has to be traversed in search of a free *Mem-*

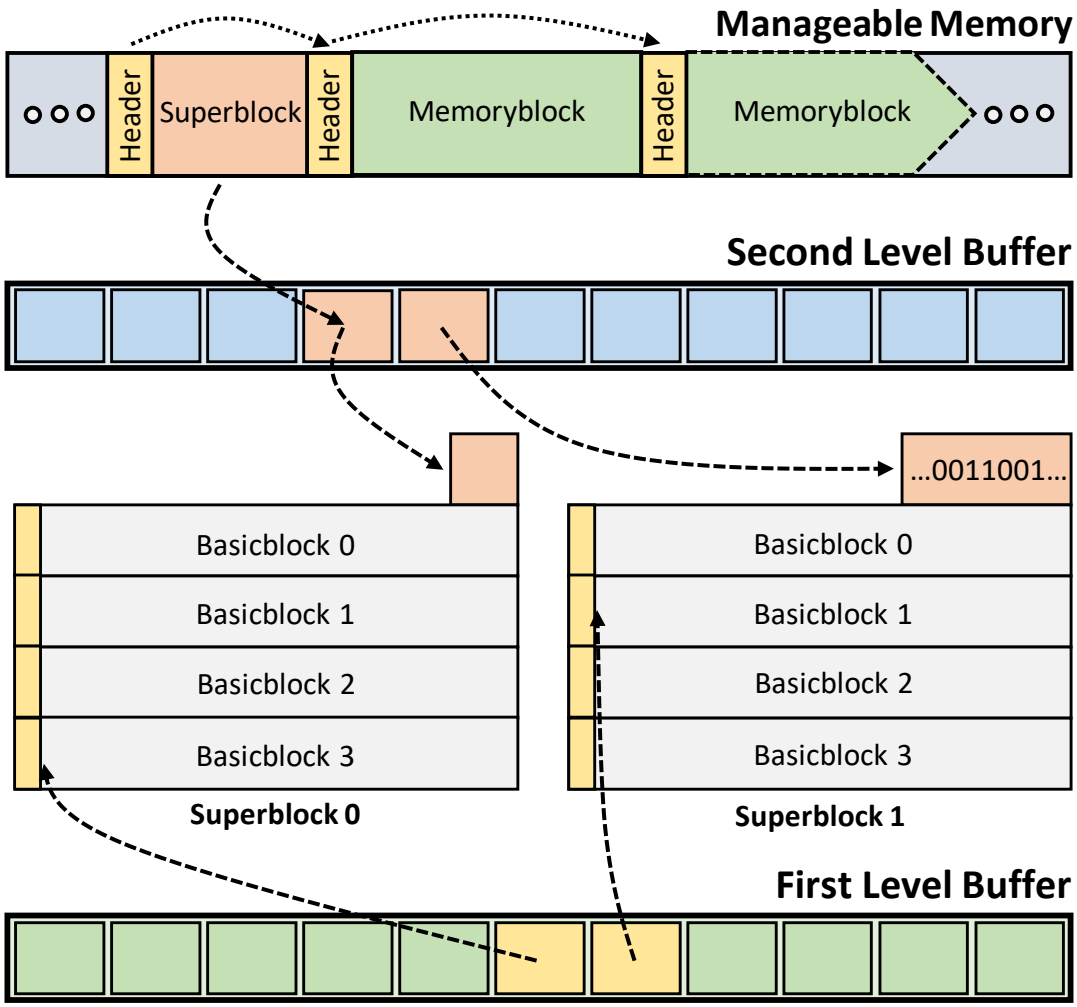


Figure 2.4: Overview of memory allocation levels in *XMalloc*. The manageable memory area is split into *Superblocks*, which are further split into *Basicblocks* used for smaller allocations, as well as larger *Memoryblocks*. Two fast lock-free FIFO queues are used as first points of contact for better performance.

oryblock. Small allocations are rounded to a statically determined size and are preferably allocated from a free list (one per static size) that holds previously allocated memory areas, called *Basicblocks*. *Basicblocks* (referenced from the *first level buffer*) are allocated from *Superblocks* (referenced in the *second level buffer*). One *Superblock* is split into 32 *Basicblocks*. Both buffers are fixed-capacity, lock-free FIFO arrays, implemented with SIMD-width coalescing. If a free-list is empty, it is refilled from buffered *Superblocks*. New *Superblocks* are only allocated if the second level buffer is also empty. Deallocation varies on the different levels. Within a *Basicblock*, just header information is updated, which might increase internal fragmentation. If a *Basicblock* is free, it is put into the first

level buffer again if possible, otherwise returned to the parent *Superblock*. *Superblocks* and *MemoryBlocks* are freed by merging with neighboring free blocks of memory.

2.3.3 ScatterAlloc

ScatterAlloc [53] was introduced in 2012 and addresses the problem of collisions during allocation by scattering the allocation requests across its memory regions, favoring allocation speed over fragmentation. To guarantee correctness and avoid deadlocks, *ScatterAlloc* focuses on a mostly lock-free design. It keeps the number of data accesses low to increase memory-access performance and avoids atomic operations on the same data word when-

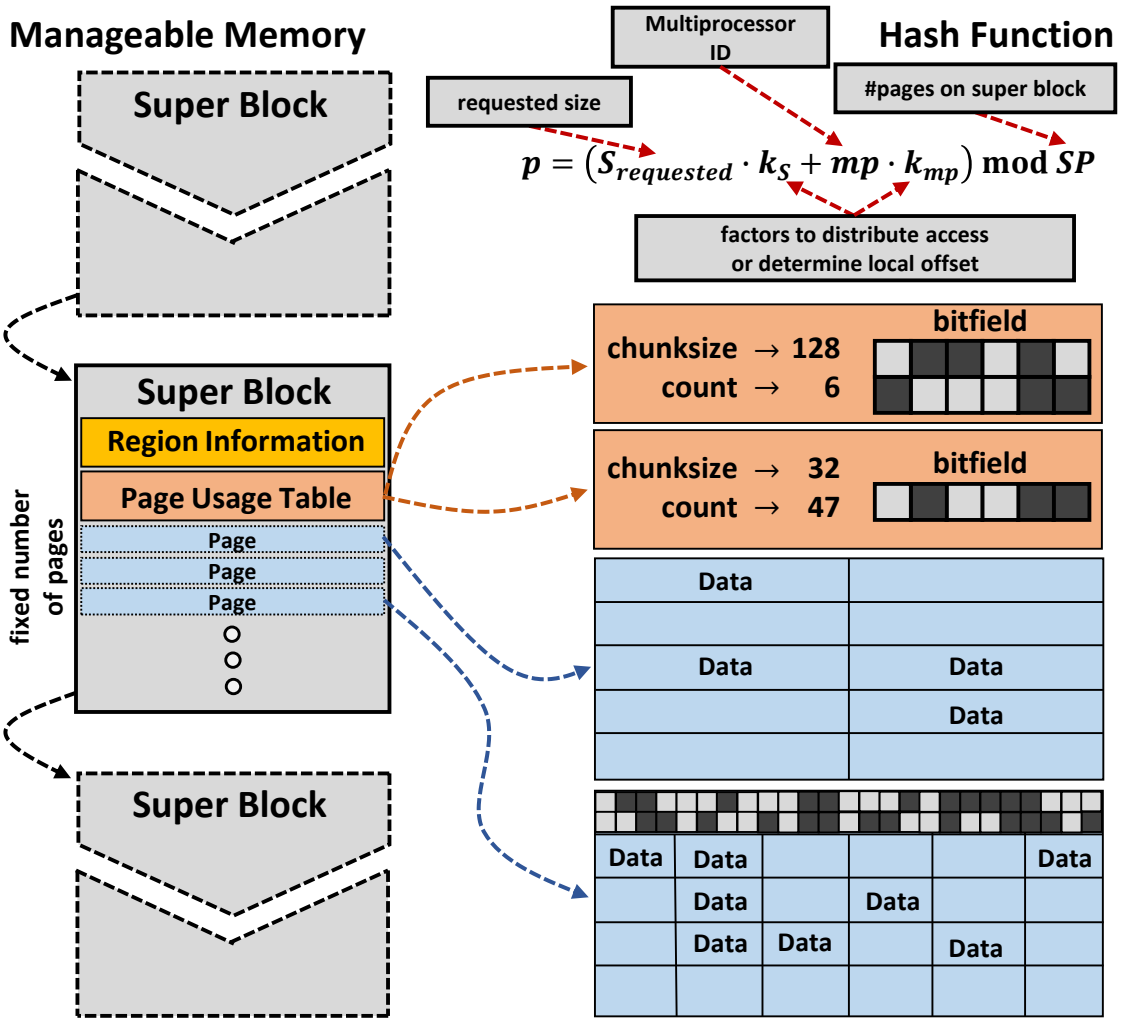


Figure 2.5: Overview of *ScatterAlloc*, which pre-splits its memory regions into *super blocks*, which are further split into fixed-size pages. Allocation state is either directly tracked on a page or in the page usage table of the *super block*. A hash function is used for increased access speed while still guaranteeing locality for allocations within on *SM*.

ever possible. Furthermore, *ScatterAlloc* also attempts to place data words close together in memory, which are allocated by threads within the same block at the same time.

Memory is split into fixed sized *pages*; free memory within a *page* is tracked via a *page usage table*. Pages are grouped into *super blocks*, which store additional meta data about their current allocation status to speed-up the allocation within a *super block*. *Super blocks* are of a fixed size and are organized in a single-linked list. *ScatterAlloc* is designed such that *super blocks* can either reside in one large region or be allocated individually, allowing for resizing of the manageable memory area. One can also pass additional memory to *ScatterAlloc*, which will then be available at the next kernel launch. Each *page* can be split into equally sized chunks, this chunk size is set at the first allocation from a *page*. *Pages* are reusable once all chunks on it have been freed again. A *page usage table* is used to track free chunks within a page, whereas each entry in this table consists of the chunk size, the number of currently allocated chunks and a bit-field with one bit per chunk to manage the allocation status. The bit-field used is 32 bit long, to support more chunks per page, a second hierarchy level is introduced on the page itself, which allows for a maximum split of 1024 chunks on a page.

To quickly find new pages for allocation, *hashing* is used. This hash function, as can be seen in Figure 2.5, tries to reduce internal fragmentation and improve cache utilization by incorporating the multiprocessor ID. In case the current page is already fully used, linear probing is used. This will still result in local clustering of chunks of the same size. There also exist two levels of meta data to speed up the search for free chunks. *ScatterAlloc* keeps a pointer to the currently active *super block*. Only once this reaches a certain fill level, the next super block in the list is investigated. A *super block* is also subdivided into equally sized regions. This also increases the search speed, as a region can be quickly reject of no suitable chunk can be found. Data requests, which do not fit onto a single page, can be served by allocating multiple, consecutive *pages* from specially reserved *super blocks*.

2.3.4 FDGMalloc

FDGMalloc [59] introduces a memory allocator with a focus on explicit warp-level programming. Their main goal is reducing branch divergence to increase SIMD scalability. By introducing a few constraints alongside such as no general *free* mechanic and the restriction of allocations only on a warp level, it reduces its applicability as a general-purpose memory manager. *FDGMalloc* organizes their design in a similar fashion to *ScatterAlloc* and *XMalloc*, by utilizing *super blocks*, which can be split into smaller chunks of memory. The main difference is that within *FDGMalloc*, one *super block* is shared by all threads within a warp. Voting is used to determine a leader thread, which does all the work, to reduce the number of simultaneous memory requests. Each warp has its own heap. All memory requests are organized through the *WarpHeader*, as can be seen in Figure 2.6. It contains a pointer to the current *super block*, as well as a pointer to a list of *super blocks* that have been allocated using the *CUDA-Allocator*. These lists are of fixed size and are

replaced once full. Each lists keeps track of how many *super blocks* are already allocated in *SB_Counter* and each *super block* tracks the number of allocations in *SB_Allocated*. Threads within a warp first determine which ones require allocation, this is once again done using a voting function. The warp header is allocated from the *CUDA-Allocator* and pointers are distributed to all participating threads. During allocation, the requested size is first aligned to a power of two, with the lowest allocation size being 16 B. If the total requested size per warp is larger than the maximum *super block* size, then the request is forwarded to the *CUDA-Allocator*. Otherwise, the associated *super block* is used to allocate the memory. If not all requests can be satisfied within this *super block*, the remaining

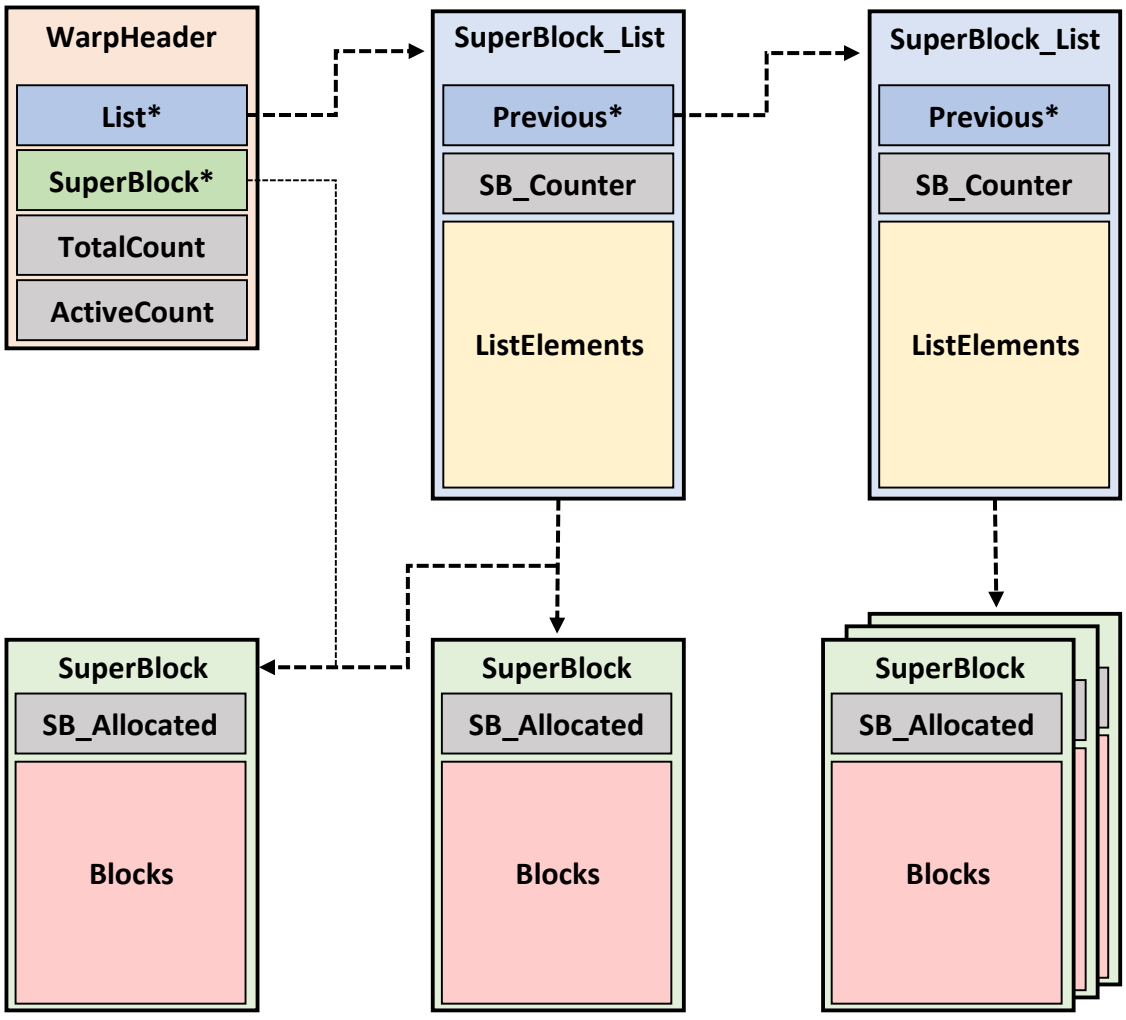


Figure 2.6: Overview of *FDGMalloc*, which organizes all allocations on a warp-level in a so-called *WarpHeader*. This holds a list of *super blocks*, from which smaller blocks can be served to the user during allocation. No general free mechanic is supported, only all allocation by a warp can be freed simultaneously.

threads will once again vote on a leader thread and start allocating a new *super block*, registering it in the *super block* list as well. A prefix-sum is used to determine the local offsets within the requested memory, served as a large block for all participating threads.

Deallocation is only possible collectively on a warp-level, there is no way to free single allocations. Furthermore, to make allocated memory available for successive kernel launches, a pointer to the *WarpHeader* has to be stored in global memory. This can lead to traversing all lists a freeing the allocated memory can be done following one of three strategies:

- Deallocating all memory is done by traversing all lists and freeing all previously requested memory pointers, leaving an intact *WarpHeader* with one remaining *super block*
- Freeing all memory including the warp header, this is intended to be called at the end of a kernel to free all memory
- To make the allocated memory available for successive kernel launches, a pointer to the *WarpHeader* has to be stored in global memory

All strategies manipulate the *ActiveCount*, reducing it to zero leads the executing thread to run one of the three strategies.

All in all, *FDGMalloc* presents a warp-level optimized approach to dynamic memory allocation with constraints that do not fit many modern applications, especially focusing on the independent thread scheduling behavior present on NVIDIA GPUs since Volta.

2.3.5 Register Efficient Memory Allocator for GPUs

Vinkler and Havran [56] propose a dynamic memory allocator based on a circular memory pool organized as a single-linked list. Variants of this will henceforth be called *RegEff*. First, they propose a very simple allocator based on *atomically* increasing an offset into a large buffer with a wrap-around once the end has been reached. This can be used for very simple dynamic memory needs, but is not classed as a general purpose memory manager, as memory cannot be freed and will be overwritten after a wrap-around. The linked list approach is simpler compared to *XMalloc*, as only one level of allocations and no caching with buffers is used.

Each allocated chunk of memory also carries header information.To not serialize allocation from a large, initial block in the beginning, *RegEff* pre-splits the memory into many chunks, similar to *ScatterAlloc*, except that these chunks need not be uniform in size. This splitting procedure generates a structure similar to a *binary heap*, as can be seen in Figure 2.7. The memory not used by the heap forms the last chunk.

During allocation, *RegEff* tries to locate the first free chunk large enough to hold the allocation, by starting from the current *memory offset*, which is stored in a shared variable. *Atomic Compare-And-Swap* is used to try allocating a chunk. If a free chunk is

large enough according to a maximum fragmentation constant, it is split into two chunks during allocation. After successfully creating the header data for the new chunk, the *memory offset* is updated to the following chunk.

Deallocation might require merging two neighboring chunks. This entails trying to allocate the next chunk such that it cannot be used by another thread. After that, the corresponding header information for the newly merged chunk is updated.

This design is called *CircularMalloc* (*RegEff-C*), and based on this, three further variants are proposed. *Circular Fused Malloc* (*RegEff-CF*) fuses the two header words into one if less than 2^{31} allocations are expected. *Circular Multi Malloc* (*RegEff-CM*) and *Circular*

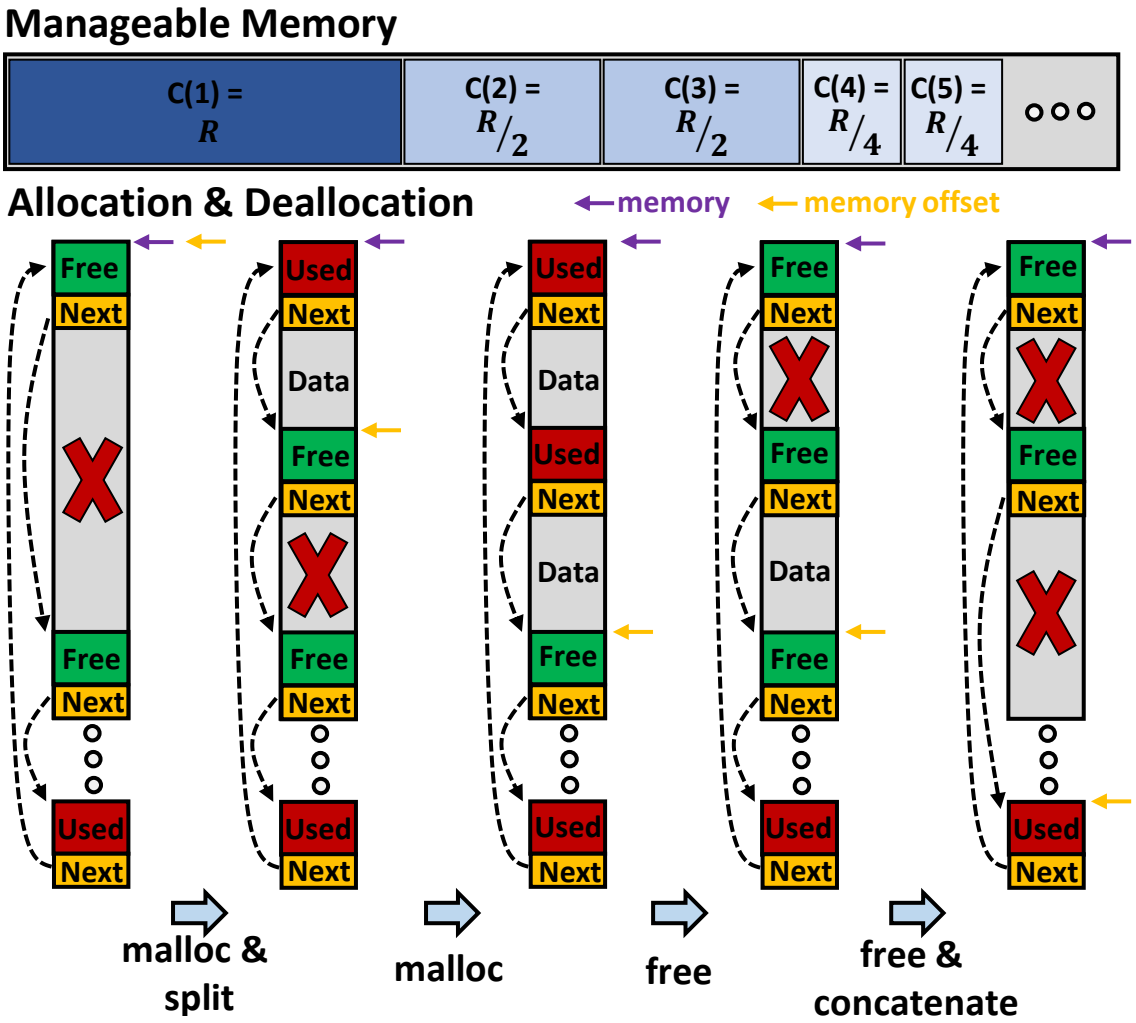


Figure 2.7: Overview of *RegEff*, which pre-splits its manageable memory region into a circular memory pool, organized as a single-linked list and structured like a binary tree. Allocations can split the next block or take an already fitting block, while deallocation typically just updates flags, but can also merge neighboring blocks.

Fused Multi Malloc (*RegEff-CFM*) trade fragmentation for speed by introducing an array of offsets (on for each SM) instead of having just one shared *memory offset*. This decreases the number of atomic collisions at an increased rate of fragmentation. Additionally, the size of the pre-split chunks is changed by dividing by the number of SMs, located by the array of offsets. These smaller heaps are linked together once again into a single-linked list.

2.3.6 Halloc: a high-throughput dynamic memory allocator for GPGPU architectures

Halloc [1] was presented in 2014 at the GPU Technology Conference. The system starts by allocating *slabs* of 2-8 MB in its initialization phase, which can then be assigned to an allocation size at runtime. The core of *Halloc* is a *bitmap heap*, which has one bit for each chunk or block that can be allocated from the system. To allocate a free block, a hash function, as noted in Figure 2.8, is used to traverse the corresponding bitmap. This visits all blocks and is fast and scalable, as long as $\leq$85 % of the blocks are allocated. Incrementing the counters is done using a *warp-aggregated atomic increment*. This selects a leader within a warp and only the leader increments and broadcasts the results to the threads in their group(up to 32$\times$ less atomics). After that, a corresponding *slab* is located and a free *block* is searched for using *hashing*. If no block was found, a new *slab* must be found and the head is moved to this *slab*. This head replacement can affect performance severely, hence *Halloc* assigns *slabs* to classes. *Free slabs* can switch between chunk sizes, *sparse slabs* ($\leq$2 %) can switch between block sizes within the same chunk and *busy slabs* (>60 %) are normally not used during head search, except when no other blocks are available anymore. Head replacement starts early (fill level > 83.5 %) to reduce this impact. Deallocation first locates the corresponding *slab* for a pointer and then updates all counters. This can result in a *slabs* moving to a new class for very sparse *slabs* or in marking a *slab* as free, which takes more time.

Allocations larger than 3 KiB are relayed to the *CUDA-Allocator*, during deallocation, if no *slab* is found, it has to be a pointer from the *CUDA-Allocator*.

2.3.7 KMA: A Dynamic Memory Manager for OpenCL

KMA [49] is the first dynamic memory allocator for *OpenCL* [28] and is designed as a two-layer memory manager, providing basic `malloc` and `free` functionality as well a higher layer built for managing dynamic data structures and a prototype of a top layer data structure inspired by Java's *ArrayList*. The *low-level memory allocator* provides an abstraction on top of the memory capabilities offered by *OpenCL* and implements a heap using an *OpenCL* read-write buffer. The size of the heap is fixed and cannot be changed at run-time and has to be initialized first by means of a special kernel which sets up the initial state and all the required data structures. The structure of the heap can be see in Figure 2.9. It consists of a state header, keeping track of a number of so-called

superblocks, which are further split into smaller blocks (splits can be any power of two) to serve to the user. A bitmap on each *Superblock* tracks the allocation state of these blocks. Free *Superblocks* are tracked via a "free"-list (implemented as a lock-free queue [35]), merging neighboring blocks is not supported, hence the largest size that can be allocated is determined by the *Superblock* size.

To allocate a block, first a corresponding *Superblock* is located by querying the *Superblock* hashmap (`sb` in Figure 2.9). If no appropriate *Superblock* is found, one is taken from the free-list and is initialized. Reserving a block on a *Superblock* can be done with an atomic operation on the *Superblock* state, which reserves a block. Lastly, by iterating

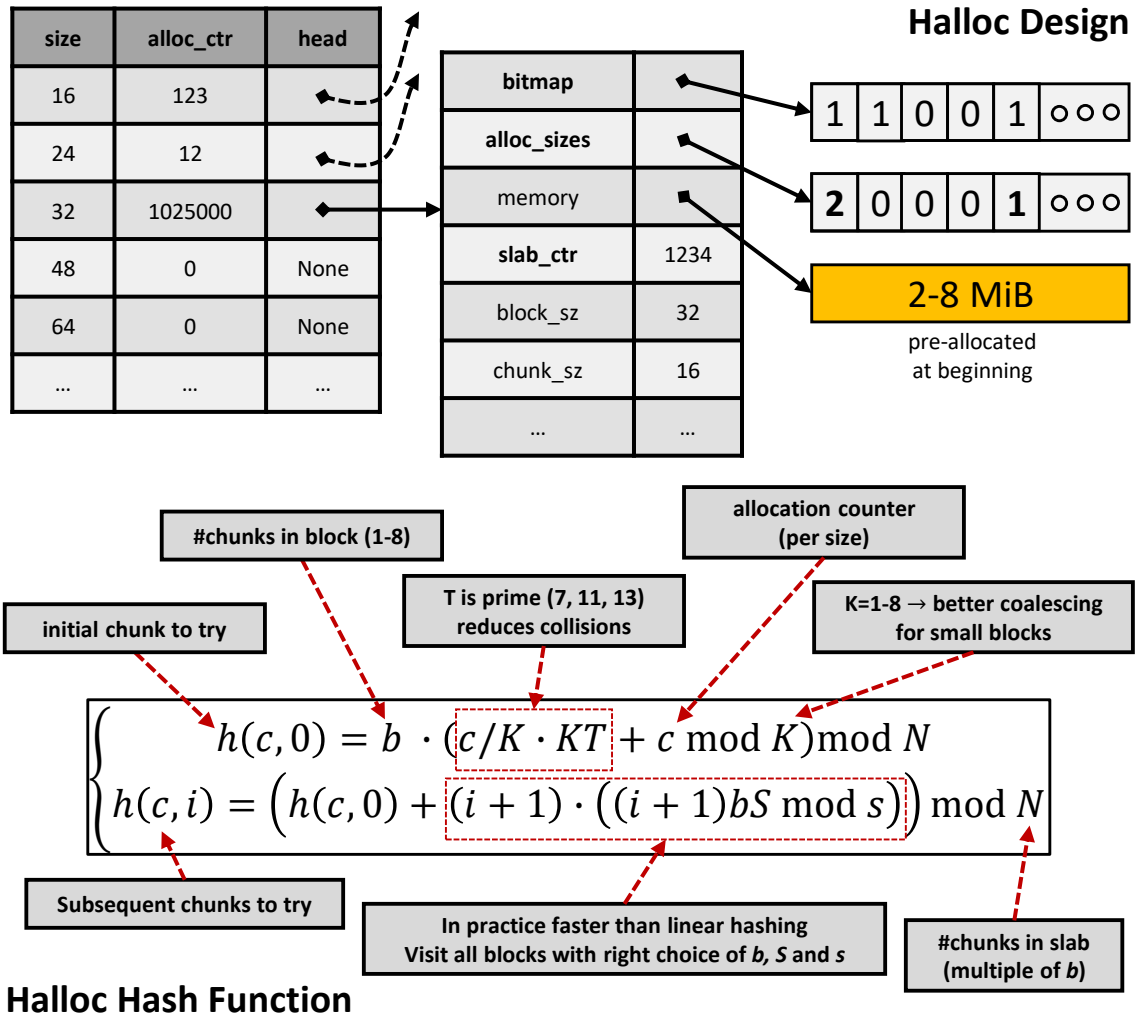


Figure 2.8: Overview of *Halloc*, which allocates slabs during its initialization phase, which can be assigned to a certain size at runtime. The *current* slab is tracked via a head pointer and replaced once considered busy (reached a fill-level of 85 %). Allocations on a slab are done using a hash function.

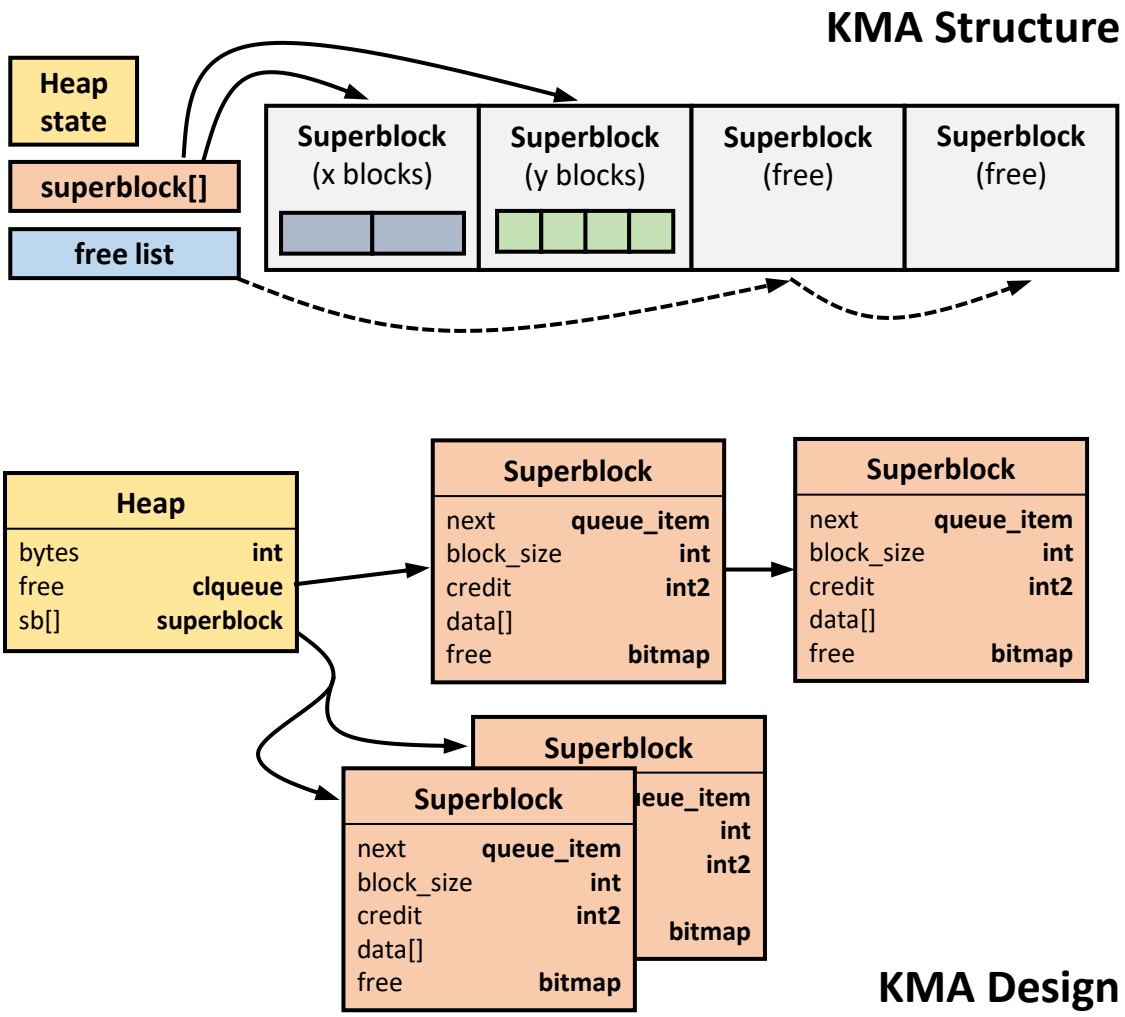


Figure 2.9: Overview of *KMA*, which pre-splits its heap into *superblocks*, which are further split into power-of-two blocks served to the user. Blocks are located via a look-up using a hash map. Free *superblocks* are tracked in free-list implemented as a lock-free queue.

over the bitmap on the *Superblock*, a free block can be located.

Deallocation reverses this procedure. First, the state variable on the *Superblock* is updated, then the bitmap entry is set and lastly, if the *Superblock* is now completely empty, it is inserted into the free-list as well.

2.3.8 DynaSOAr: A Parallel Memory Allocator for Object-Oriented Programming on GPUs with Efficient Memory Access

DynaSOAr [50] deals with the problem inherent with object-oriented programming on the GPU, which is the suboptimal memory layout once objects are laid out in memory as an *Array of Structures (AOS)*. They propose a fully-parallel, lock-free dynamic memory

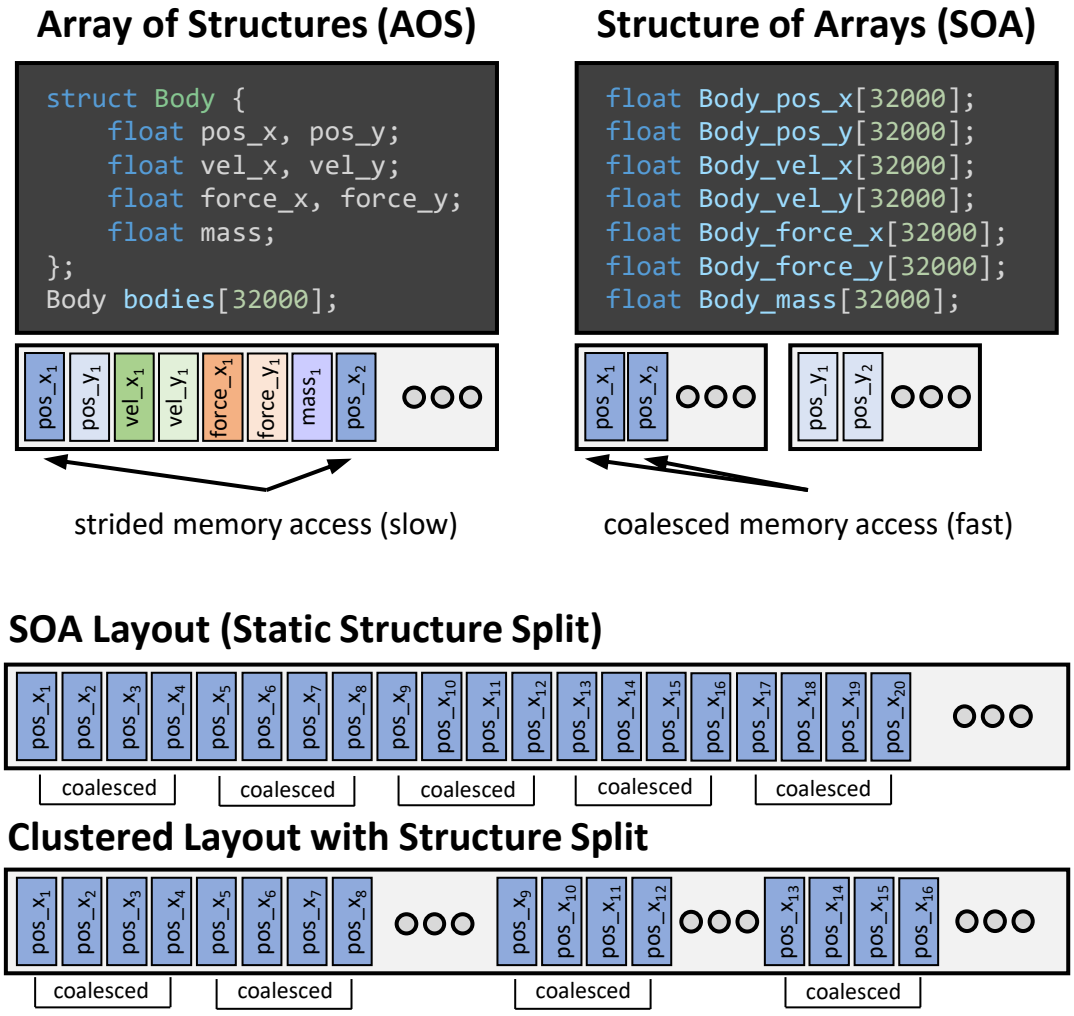


Figure 2.10: *DynaSOAr* deals with the problem inherent between *AOS* and *SOA* layouts for data storage on the GPU. To realize a dynamic setting, which does not work well with a static split, *DynaSOAr* uses a clustered storage format with a structure split.

allocator, a DSL-style data layout as well as a *do-all* operation on this data. This essentially lays objects out in a *Structure of Arrays (SOA)* layout (see Figure 2.10), drastically improving memory access performance, trading memory access speed for allocation speed. A straightforward *SOA* layout achieves great memory access performance, but is static in size. *DynaSOAr* bases its design on the insight that splitting the original layout into a *clustered layout* has the same overall cache/vector performance, if the cluster size is at least the size of a cacheline, i.e., 128 B. Memory fragmentation that might be caused by the allocation and deallocation of dynamic objects is minimized by utilizing a *hierarchical bitmap*. Further improvements include the coalescing of memory requests on a warp-level to reduce the number of overall allocations.

Due to this internal data layout and requirement of only pre-defined structures being managed, it cannot be used as a general-purpose memory manager.

2.3.9 Throughput-oriented GPU Memory Allocation

BulkAllocator [20] introduces the bulk-semaphore, a throughput-oriented synchronization primitive, as well as two allocators. The bulk semaphore enables pre-emptive batch allocation, combating *false-resource starvation* and the *scalability barrier*. Building on three

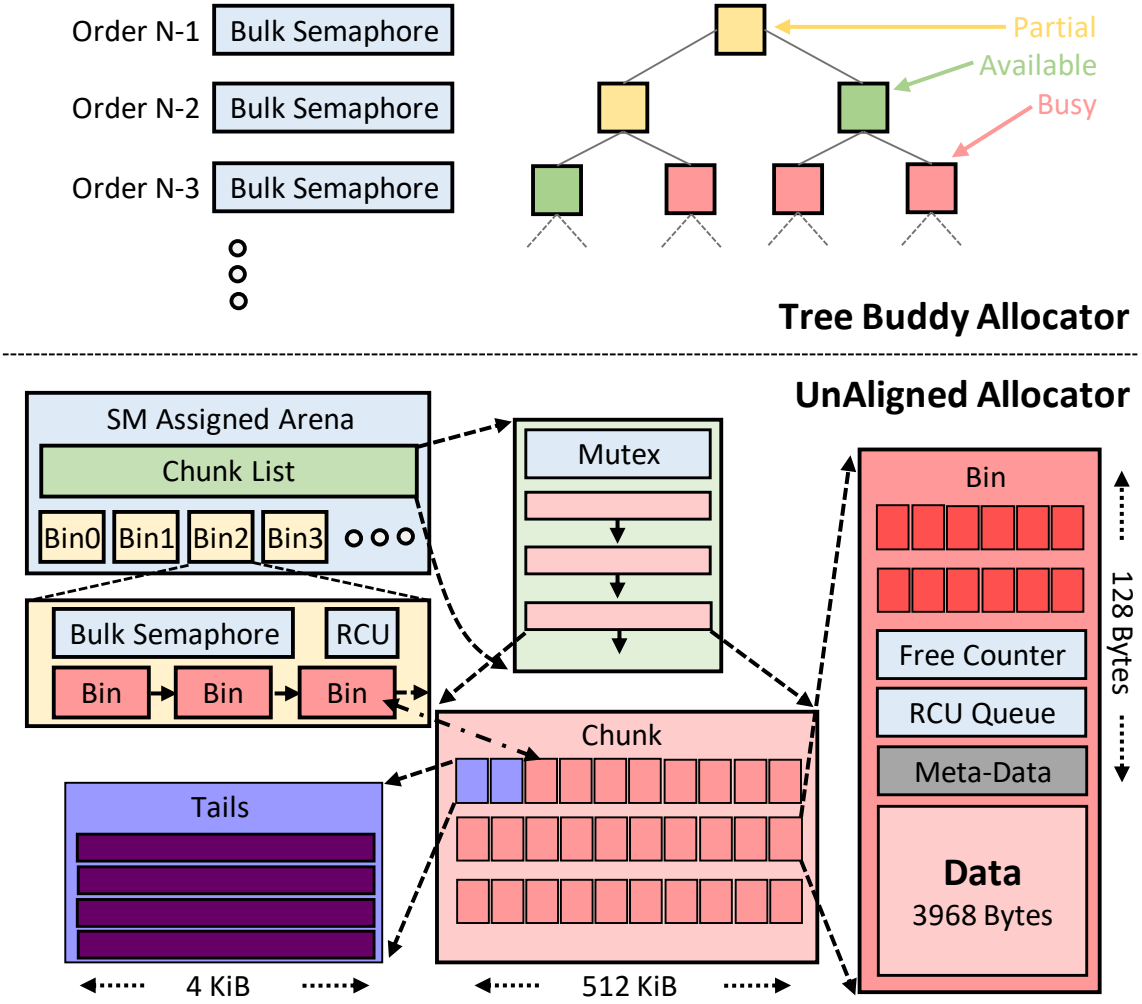


Figure 2.11: Overview of the *BulkAllocator*, which consists of two allocator designs using the *bulk semaphore*, a synchronization primitive for batch-allocation. The *Tree Buddy Allocator* (used for larger allocations) is implemented as a static binary tree with one *bulk semaphore* per level and the *UnAligned Allocator* is based around a common CPU memory manager design, with one memory arena per *SM*, per-size bin lists secured by a *bulk semaphore* and allocates its chunks from the *Tree Buddy Allocator*.

counters, threads can determine if the current allocation routines will also satisfy their own demands. If that is not the case, they can start immediately with a new bulk allocation, significantly reducing waiting times, especially during high access pressure.

This is a crucial part of both allocators. The Unaligned Allocator (*UAlloc*), which closely resembles existing concurrent CPU allocators, is used for all allocations smaller than 2 KiB. A Tree Buddy Allocator (*TBuddy*) is used for all allocations larger than that. The bulk-semaphore is used throughout as the synchronization primitive. *TBuddy* is modelled as a static binary tree, tracking the state of large memory blocks. Each level in the tree is secured by a *bulk semaphore*, each node can be either **busy**, **partial** or **available**. Node status changes are propagated from node to parent. To ensure consistency, both node and parent are locked. *UAlloc* uses one memory arena per SM, handling chunks of 512 KiB which are further sub-divided into 4 KiB bins (static size per bin). Each arena keeps a per-size bin list of bins with available elements. The first two bins (called *tails*) in a chunk keep track of the allocation state of the bins within a chunk. New bins are allocated from a chunk in the chunk list, new chunks are allocated by using *TBuddy*. To update the bin free-list, they use Read-Copy-Update [32] as their synchronization mechanism.

They test allocation sizes between 8 B and 512 KiB, reporting increased performance over the *CUDA-Allocator* for all tested allocation sizes except for 2 KiB, 4 KiB, 64 KiB and 128 KiB. Unfortunately, no public version is available for further testing.

2.3.10 Queues

Many of the aforementioned allocators use queues to store available chunks/pages/bins, for various sizes, thus efficient queuing is important. Efficient parallel queue management is a long standing research topic. For example, a parallel array-based queue similar to current GPU designs has been proposed by Gottlieb [21] in 1983. Parallel CPU-designs have long focused on non-blocking linked-lists, including the famous Michael-Scott queue [35] and the Shann-Huang-Chen queue [16]. With larger parallelism, the ordering between elements enqueued concurrently is practically irrelevant, which can be exploited by putting elements into the same bucket [24].

Specialized GPU queue designs are scarce. Task-based GPU runtime systems have used proprietary solutions, mixing linked-lists and array-based queues [51, 52]. Scogland and Feng proposed a blocking array-based GPU queue [45]. Unfortunately, their queue blocks on empty states making it impracticable for memory allocation. The BrokerQueue [26] removes these blocking states by pairing concurrent enqueues and dequeues. Blocking only happens to ensure ordering between pairs. While link-based queues like Michael-Scott or Shann-Huang-Chen also work on the GPU, their performance is multiple orders of magnitude lower than array-based designs [26]. Thus, even Gottlieb's original work performs significantly better.

While array-based queues are thus preferable, all previous designs use a fixed size ring-buffer for efficient access. In conjunction with per-bin size queueing and unpredictable

memory allocation requirements, statically-sized, array-based queues lose their attractiveness for dynamic memory allocation as they significantly increase memory requirements.

With *Ouroboros* and *ouroGraph*, we combine the best of both strategies. It is as efficient as an array-based queue, but is completely built on dynamic memory.

2.4 Graph Algorithms

While a dynamic graph representation is certainly an important goal, algorithmic performance on top of the representation is also important. Many standard graph algorithms have been implemented on the GPU. These include triangle counting [23, 43], which can be used to find key players in a network based on their local connectivity; PageRank [12], which measures the importance of web pages according to the links to a page, connected components [47], single-source shortest path [17], the betweenness centrality [33, 44], graph clustering and connectivity [7], and community detection [48]. To display the suitability of our designs for memory-intensive algorithms, we test the performance for *PageRank* and *static triangle counting (STC)*, the first exemplifying a simple adjacency traversal while the second involves significant traversal overhead and edge look-ups leading to irregular memory access patterns and divergence. Furthermore, both are also available in both *cuSTINGER* and *Hornet* and enable a comparison to these different storage layouts.

3

aimGraph - Autonomous, Independent Management of Dynamic Graphs on GPUs

Contents

3.1 Introduction

Previous attempts at dynamic graph management on the GPU still required a tight bond with the CPU for reallocation tasks. This not only introduces additional overhead caused by synchronization with the CPU, but also the need to split up update procedures into multiple kernels. If the allocation state for an adjacency did not suffice for the given number of updates, this adjacency would be marked as in need of reallocation. Then, the CPU would need to allocate a new block, large enough to manage the new size, copy over the existing data from the old adjacency and then launch a subsequent kernel to finalize the insertion. Deallocations would just mark edges as deleted, as freeing memory would be even more costly regarding performance, but this strategy does not bode well for memory fragmentation.

This tight link to the CPU and overly complicated update procedure puts severe restrictions on algorithms and frameworks using this data structure. This raises the question, if it might be possible to move the whole data structure as well as all management tasks directly to the GPU. This way, no synchronization with the host would be required and update procedures would also be able to finish within one kernel launch. To achieve this,

we present *aimGraph*, managing adjacencies on fixed-size edge blocks, linked together for larger adjacencies and managed by a memory manager directly on the GPU. The next sections detail the ideas behind and design of *aimGraph*.

3.2 aimGraph

In the following section we sketch the design and general idea behind *aimGraph* (**a**utonomous, **i**ndependent **m**anagement of dynamic **Graphs** on the GPU) and focus on the initial memory setup and layout, the update implementations as well as performance relevant optimizations. This section is followed by a comparison to *cuSTINGER* and a performance evaluation.

3.2.1 Memory Layout

aimGraph initializes the system with a single GPU memory allocation, assigning a large block of memory (this is currently set to take as much memory as is available on the device) to the framework. All following allocation calls are handled internally by requesting memory from a simple memory manager. This memory manager is initialized from the CPU with a number of fixed parameters (setting up the edge mode, the block size, kernel launch parameters) and then placed at the beginning of the large block of memory previously allocated on the device. It holds a pointer to the beginning and end of the allocated device memory and also stores all the necessary management data. Using this CPU-autonomous memory management approach, the framework can facilitate all dynamic memory needs directly on the GPU and significantly reduce the run time overhead by avoiding individual allocation calls from the host. Especially considering changes to a large number of adjacencies, which would otherwise have to be reallocated sequentially on

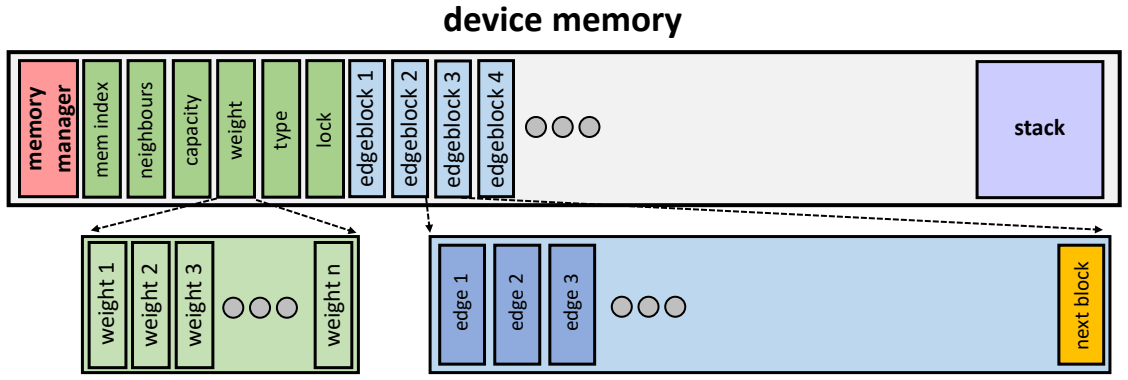


Figure 3.1: Visualization of the device memory layout as managed by the *memory manager*, which is placed at the beginning of this region. Vertices are placed aligned right after the memory manager, followed by the adjacency data stored on edge blocks. The end of the manageable memory is usable as a stack for temporary data.

the host, this significantly improves update performance.

Our memory manager follows a similar approach as traditional memory management in a CPU C/C++ program. Static data is placed at the bottom, the dynamic "heap" area is placed right after the static data, while the temporary data on the stack grows from the top down. We consider these three regions in our memory manager on the GPU. The memory manager holds a global edge block index, which denotes the currently largest edge block index allocated within the system. To allocate a new edge block, an allocating thread has to increment this index *atomically*. Hence, dynamic allocation in the system can be done lock-free.

3.2.1.1 Static data

Similarly to *cuSTINGER*, the number of vertices is considered static in our implementation. Adding or deleting vertices from the graph is not supported[1], hence the size of the static data segment is known at the time of initialization. As previously mentioned, right at the start of the application, the **memory manager** and, after that, vertex management data structures are placed in device memory. The management data is set up as a structure of arrays (*SOA*), each array the size of `numberVertices` · `sizeof(parameter)`. The parameters in question are

- `memindex`: Indicates the start of the adjacency data per vertex via the block index of the first dynamic edge block. As the edge block size is fixed, a single, 32 bit index suffices to address all potential edge blocks for any reasonable allocation size[2].
- `neighbors`: Number of neighbors in the adjacency
- `capacity`: Maximum number of neighbors in the adjacency with the current block allocation, i.e., `neighbors` $\leq$ `capacity`
- `weight`: A weight can be assigned to each vertex `<optional>`
- `type`: A type can be assigned to each vertex `<optional>`
- `lock`: Algorithms can restrict access to individual vertices using this lock

All individual arrays are placed cacheline size aligned and each is at least the size of a multiple of the general GPU cacheline size of *128 bytes*. Each vertex requires at least $4 \cdot 4\,\text{B} = 16\,\text{B}$ (up to 24 B when using both types and weights).

[1] Dynamic vertices are first introduced with *faimGraph* in Section 4

[2] Given an edge block size of 64 B, 257 GB of useable adjacency storage can be accessed, which can support in the best case $\geq 2^{35}$ edges of a simple graph

3.2.1.2 Dynamic data

The adjacency data after the static vertex segment is managed in blocks, the block size depends on the application and the size of the edge data. Each block stores adjacency data and uses the last 4 B to indicate the location of the following edge block. For a simple adjacency storing just the destination vertex, 64 B suffices for most scenarios, i.e., has room for up to 15 edges on one edge block. For semantic graphs, the block size is larger to accommodate more edges per block. Additionally, different update mechanisms profit from different edge block sizes, depending on the update strategy and the average size of the adjacency per vertex an optimal block size for the given graph is chosen.

The initialization works fully in parallel, contrary to *cuSTINGER*. In a pre-computation step, an `exclusive prefix scan` is used to determine the memory requirements for each vertex, using the parallelism at hand to the maximum extent. The last element in an edge block is always an index to the next edge block. This makes this approach a combination of a linked list and an adjacency array and allows for memory locality for vertices within an array. Depending on the processing model, this enables coalesced memory access on an edge block. At the same time, this strategy avoids reallocation of the whole block if augmentation is required, as another edge block can be allocated by simply updating the index at the end of the last block. This approach also leaves the possibility to switch to more sophisticated memory management in the future.

3.2.1.3 Temporary data

In the initialization phase, but also for updating the graph and algorithms running on the graph, additional, temporary data may be required. This can, e.g., be edge updates (consisting of source and destination vertex data) or an array holding the triangle count per vertex to calculate the overall triangle count within a graph structure.

This data is managed like a stack. The memory manager holds a stack pointer pointing to the end of the allocated memory and can deal out shares of this memory to algorithms or for pushing updates to the graph structure. This way the whole device memory can be managed without considering a trade-off between the managed memory portion of the device memory and the temporary data needed. The memory manager just has to check if temporary data does not protrude into the dynamic data segment.

3.2.2 Initialization

At the start of the application, a graph is parsed into an intermediary *CSR* (Compressed Sparse Row) format and in the beginning, a pre-processing kernel is started to calculate the memory requirements per vertex. This kernel computes in detail the number of `neighbors` and from that the `capacity` and `block requirements` per vertex in parallel. Using the `block requirements` and an `exclusive prefix sum scan`, the overall memory offsets for all individual edge block lists can be computed.

Using all this information, the initialization kernel can be run completely in parallel without regard for locking, while the *CSR* format is transferred into the *aimGraph* format, only a single instruction is performed by a single (the last) thread, as this one can set the `next_free_block_index` in the memory manager, which is required for dynamic memory allocation for edge updates.

3.2.3 Edge Types

aimGraph supports three different edge types:

- **Simple**: This mode stores the bare graph structure using just the destination vertex in the edge data array
- **Weights**: This mode adds the support for weights for both vertices and edges to the simple mode
- **Semantic**: In addition to weights, this mode adds support for type information for both vertices and edges and also two timestamps per edge, which increases the size required per edge significantly

This variety of options is implemented using templated classes and methods, as most functionality is independent of the concrete representation of the edges themselves, just the modification functionality is realized via overloaded functions. Depending on the use case, one of these more advanced modes can be selected at the cost of an increased memory footprint, choosing a larger sized edge type also increases the basic block size to accommodate a larger number of edges per block.

3.2.4 Edge Insertion

Edge updates in the current setup require a single lock per vertex to combat concurrent read/writes to the adjacency, neighbors and capacity as shown in Algorithm 1. Access to the `memory manager` on the other hand simply requires *atomic memory access* to get a new block, if the current capacity cannot accommodate the new update.

This results in a high update rate, if the edge updates do not particularly favor a small set of vertices over the majority. For edge updates that are close to a uniform random distribution, inserting 1.000.000 edges can be achieved in a few milliseconds. Depending on the average size of the adjacency, even when accessing the memory manager heavily to adjust the size of individual edge block lists, performance still remains high overall.

We provide two implementations, optimized for different adjacency list counts. The first, which is the standard insertion mode, is shown in listing 1. It completes each individual update using a single thread. This approach is especially fast for small to medium sized adjacency lists (less than 50 vertices per adjacency on average).

If the average size per adjacency grows larger, the traversal of the graph structure becomes the bottleneck. Thus, for larger adjacency list sizes, we use an entire warp (32

Algorithm 1: Edge insertion using locking

```
   Data: edge update batch
   Result: Edges inserted into graph
1  Edge updates put onto stack;
2  while lock acquired do
3      read neighbours & capacity;
4      for vertices v in adjacency do
5          if v == DELETIONMARKER or index ≥ neighbours then
6              remember index;
7              break;
8          if v == edge update then
9              found duplicate, ignore;
10             break;
11         advance in EdgeBlockList;
12     if !edgeInserted and!duplicateFound then
13         get memBlock from memManager;
14         update adjacency, index, capacity & neighbours;
15     else if !duplicateFound then
16         insert element at index;
17     release lock;
```

threads) for the update. In this case the *for-Loop* reduces to a loop over blocks and the memory access pattern within blocks can be optimized to requesting a full *cacheline* per warp at once.

3.2.5 Edge Deletion

Edge deletion works in a similar manner to edge insertion, the major difference results in the fact that there is no need to access the memory manager, as no new memory will be required. Additionally, we do not return empty blocks to the memory manager, but simply reuse them when edges are inserted for the same node again. In this way, we can avoid access to the memory manager completely during deletion.

As with the insertion process, two different implementations are provided, one launching a single thread per update and the other launching a full warp per update, depending on the average size of the adjacency. In the following Algorithm 2, we show the deletion process for the standard launch. When launching a full warp per update, the *for-Loop* is again reduced and leads to better memory locality as a whole *cacheline* is fetched by the warp.

Algorithm 2: Edge deletion without locking

```
Data: edge update batch
Result: Edges deleted from graph
Edge updates put onto stack;
read capacity;
for vertices v in adjacency do
    if v == edge update then
        atomically update Adjacency & neighbours;
        one thread decreases neighbours;
        break;
    advance in EdgeBlockList;
```

3.3 Comparison to *cuSTINGER*

This section provides a comparison between *aimGraph* and *cuSTINGER* by investigating the respective memory footprints and composition, as well as the time spent initializing and updating the graph structure and is followed by an evaluation of the performance differences.

3.3.1 Memory footprint

One of the biggest differences stems from the way memory allocation is performed in general. *cuSTINGER* performs individual calls to `cudaMalloc()` from the CPU to allocate the management data and all individual edge blocks. Especially for graphs with more than a million vertices this results in a significant overhead, compared to the single allocation in *aimGraph*. Another big difference lies in the memory footprint. *cuSTINGER* uses pointers to

- locate attributes
- point to individual edge blocks
- point to data members within an edge block (especially prevalent in *semantic* mode)

This increases the size of the management data set and also requires a full block (of 64 B) just to hold member pointers.

aimGraph on the other hand uses an indexing system (reducing the size per pointer/index from 8 B to 4 B), but also eliminates the member pointers and additional attribute pointers by combining an efficient indexing scheme and reinterpreting memory on the fly using `casts` to achieve the same functionality at a fraction of the memory.

3.3.2 Initialization

As previously mentioned, *aimGraph* performs a single device memory allocation and can perform the whole setup in parallel on the GPU with little overhead. Compared to that, *cuSTINGER* needs to allocate each individual edge block array from the CPU, also performing the pre-computation entirely on the CPU and only the actual writing of the adjacency data per vertex occurs on the GPU in parallel. However, as there is no offset-indexing scheme, even this launch cannot utilize the GPU to its full potential, leading to an enormous performance difference in the initialization stage.

3.3.3 Updates

Here once again, the different strategy in allocating memory pays off for *aimGraph*, as updates can be achieved in a single kernel launch with a single lock per vertex when inserting edges and even without a lock in the deletion process.

cuSTINGER, on the other hand, launches at least one kernel, which cannot utilize the whole GPU due to the lack of locking, but also incurs a heavy penalty if duplicates are present or reallocation is required. In this case, new space must be allocated using `cudaMalloc()` and the whole edge block array of the given vertex is copied over. In the worst case, five kernel launches are required to deal with all eventualities, leading to significantly lower update rates.

cuSTINGER holds a slight edge in case no duplicates are in the batch, no reallocation is necessary and the average size of an adjacency is large (greater than 50), as only few operations are performed. However, in these cases there is a chance to produce invalid graphs, as there is no locking or contention resolution mechanism in place. Depending on the actual behavior of the hardware thread scheduler, this problem may show up more or less often.

Name	Network Type	—V—	—E—	Initialization (ms) *aimGraph*	Initialization (ms) *cuSTINGER*
Luxembourg	Road	115k	239k	3.13	110.5
coAuthorsDBLP	Citation	299k	1.95M	6.687	289.6
ldoor	Matrix	952k	45.57M	53.704	1 053.2
audikw1	Matrix	943k	76.71M	86.713	1 108.6
Germany	Road	12M	24.74M	101.68	14 010.7
nlpkkt160	Matrix	8M	221.17M	228.13	out of memory

Table 3.1: Initialization time in *ms* for *aimGraph* and *cuSTINGER* for a selection of graphs from the 10th DIMACS Graph Implementation Challenge [5].

3.4 Performance

The performance measurements were conducted using a NVIDIA GTX 780 GPU (3 GB V-RAM), an Intel Core i7-3770K using 16 GB of DDR3-1600 RAM. Although this is considered consumer hardware, the goal is to show differences between *aimGraph* and *cuSTINGER*. Performance on more powerful, professional equipment is expected to be even higher. The graphs used were taken from the 10th DIMACS Graph Implementation Challenge [5] and a selection used for performance evaluation is highlighted in Table 3.1.

Both frameworks use the same testing methodology, starting with initialization, followed by the generation of random edge updates, which were subsequently added to the graph and then removed again. This is done 10 times and the results are averaged to produce the overall results. Only the calls to the initialization and update functions were measured. This whole process is repeated 10 times and averaged again, hence the performance numbers shown display the average time of 10 rounds of initialization and 100 rounds of edge insertions and deletions respectively.

3.4.1 Initialization

As shown in Table 3.1, the different memory setup procedure pays off the most in the initialization step; the highest advantage is achieved when processing a high number of vertices with a comparatively low number of edges. In this case, *aimGraph* is nearly 300 $\times$ faster. Even for a low number of vertices with a high degree the speed up achieved still reaches double digits. This can be attributed to the fact that *aimGraph* works autonomously on the GPU and can parallelize the setup process. In contrast, *cuSTINGER* performs its setup process from the host with individual initialization calls per vertex, calculating memory requirements and allocating memory from the host directly.

Additionally, *aimGraph* has a significantly lower memory footprint. Thus, larger graphs can be kept in memory compared to *cuSTINGER*, as can be seen for the sparse matrix network `nlpkkt160`.

3.4.2 Edge insertion

The first three cases in Figure 3.2 show where *aimGraph* has a clear performance advantage. This is the case if the degree per vertex is small, as in those cases the over-allocation strategy of *cuSTINGER* does not provide enough space for the insertion operations and both frameworks have to reallocate, which is much faster using *aimGraph*, as everything is done on the GPU in one kernel. *cuSTINGER* has to reallocate from the CPU and also copy over entire edge blocks.

cuSTINGER has an advantage due to their overallocation policy. *cuSTINGER* allocates 50 % more to reduce the need for reallocation later on, if there is a comparatively low number of vertices compared to the number of edges (as seen with the sparse matrices). In these cases, *cuSTINGER* achieves the updates in less time than *aimGraph*, as we actually

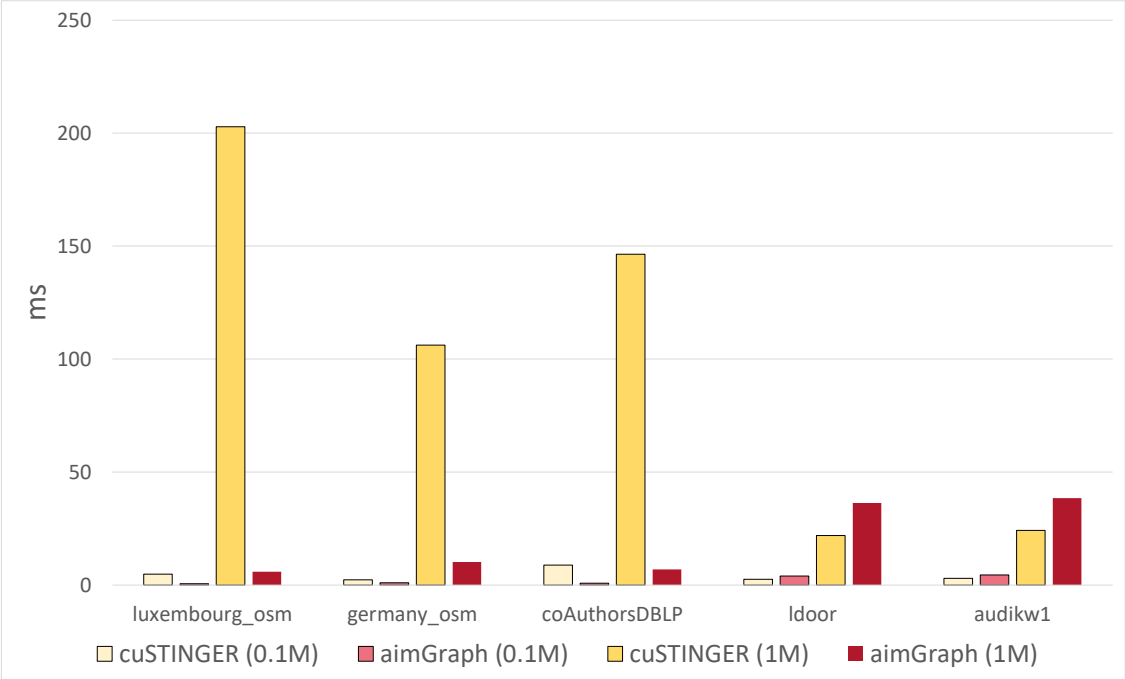


Figure 3.2: Performance measurement for edge insertions, using a batch size of 100.000 and 1.000.000. The last two cases highlight both the inherent drawback of edge block traversal within *aimGraph* as well as the huge benefit *cuSTINGER* derives from overallocation, as no reallocations have to be performed.

have to perform memory allocations, which involve more complex traversal mechanisms and locking.

Additionally, as *cuSTINGER* does not use any form of race condition avoidance, there might arise some cases that result in an invalid graph structure. Depending on GPU scheduling, duplicates within batches are not detected and remain in the graph. The behavior of *aimGraph* is independent of scheduling, and keeps a more compact memory layout. Although we employ correctness guaranties and keep memory requirements significantly lower, *cuSTINGER* only shows a slight performance advantage.

Another factor, which becomes performance relevant, is the difference in adjacency traversal. Due to the more modular structure of *aimGraph*, the traversal of individual edge lists takes longer compared to the array traversal of *cuSTINGER*, as the indexing scheme behind connecting multiple blocks into a contiguous list requires extra cycles.

For testing purposes, reducing the memory over-allocation of *cuSTINGER* decreases performance up to 100 × or changing the update strategy by first inserting 10 batches of updates and then removing them also worsens performance for *cuSTINGER* significantly, while *aimGraph* does not see a major effect on its performance. Overall, it can be noted that *cuSTINGER* only performs well when it works within its overallocation boundaries and for larger sized adjacencies. Otherwise its performance drops significantly.

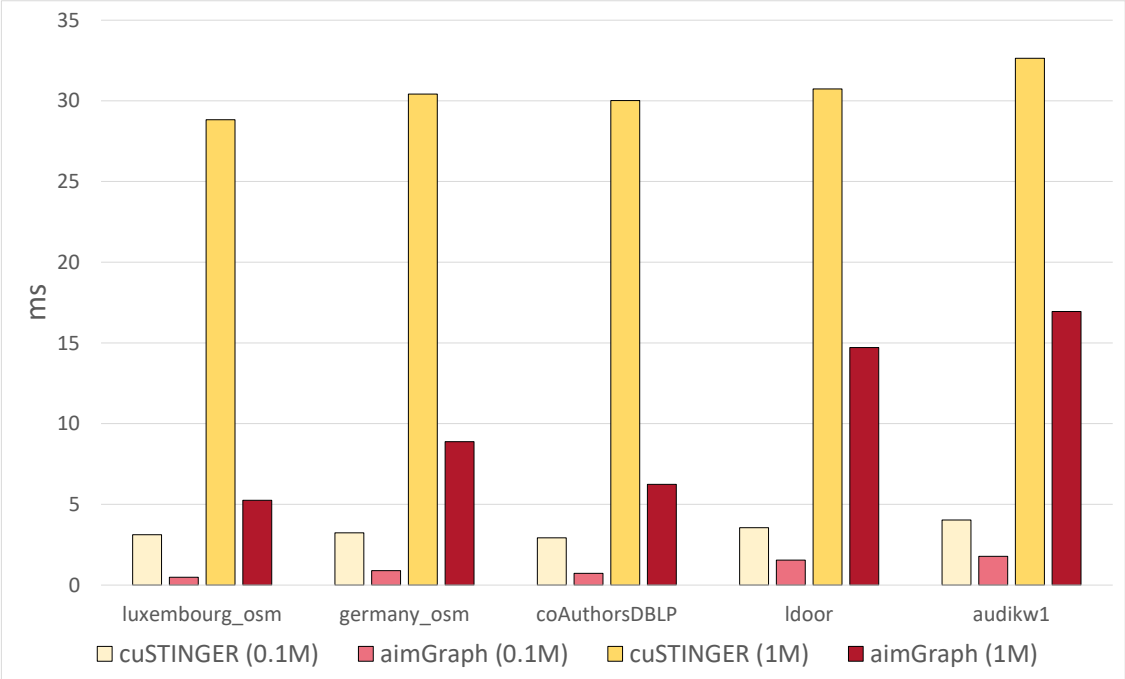


Figure 3.3: Performance measurement for edge deletions, using batch size 100.000 and 1.000.000

3.4.3 Edge Deletion

In case of deletions, the performance difference is slightly less pronounced compared to the insertion process as can be seen in Figure 3.3. This is due to the fact that deletions always work without rearranging the general memory layout and also there exists no possibility of adding/removing duplicates.

aimGraph uses variant 1 of the deletion procedures for the first four graphs (one thread per update), as the average adjacency is comparatively small to medium sized (less than 50 vertices per vertex on average). Under these circumstances adjacency traversal is less important compared to stalling threads. The performance benefit is therefore greatest for very small adjacencies per vertex and becomes less prominent for larger adjacencies.

The last case uses variant 2, launching warp-sized blocks, as in those cases the adjacency traversal is crucial to performance, and once again performance is about 2× faster compared to *cuSTINGER* for the tested graphs. The main difference to *cuSTINGER* is the single kernel launch (compared to two launches for *cuSTINGER*) and the more efficient duplicate checking.

3.5 Discussion

aimGraph is a memory-efficient streaming graph solution on the GPU that enables very high update rates without the need to transfer the graph data structure to and from the host. This solution is purpose-built for the GPU, keeping memory requirements low by using an indexing structure instead of pointers and managing the device memory on the device autonomously, without the need for copying and allocating new blocks from the host. In this way, updating the graph structure can be achieved with a single kernel call respectively and allows for concurrent initialization and updates.

The current implementation includes support for different semantic modes (including simple, weighted and semantic graphs) and offers developers different updating strategies, which can be selected for specific workloads for optimal performance. Furthermore, different verification methods are present to test and verify new features and algorithms. Even on consumer-level *GPUs* (NVIDIA GTX 780 with 3GB VRAM), the framework can hold tens of millions of vertices and hundreds of millions of edges in memory (depending on the semantic mode) and is also able to process 20–100 million insertions per second and between 50–150 million deletions per second.

Overall, *aimGraph* offers an efficient and fast dynamic graph implementation with low memory footprint and autonomous memory management, allowing for different update mechanisms tailored to different graph properties.

4

faimGraph - High Performance Management of Fully-Dynamic Graphs under Tight Memory Constraints on the GPU

Contents

4.1 Introduction

aimGraph already provides a basis upon which dynamic graph management can operate autonomously on the GPU. This removes the tight bond with the CPU, which makes repeated kernel launches and synchronization unnecessary. Compared to *cuSTINGER*, it is more memory efficient and especially initialization is vastly improved due to the ability of *aimGraph* to utilize the parallel capabilities of the GPU to its full potential. Additionally, especially for sparser graphs and significant changes to adjacencies, *aimGraph* shows greatly increased update rates over *cuSTINGER*.

But still, this initial design leaves some aspects of dynamic graph management unaddressed and a lot of optimizations on the table. Similar to *cuSTINGER*, memory cannot be re-used within the system, which might result in larger overhead during prolonged use. Edge blocks once allocated to an adjacency stay with that adjacency, which can be beneficial for deallocation and repeated flux for the same adjacencies. This might also result in earlier than necessary failure due to out-of-memory, especially if graphs show lots

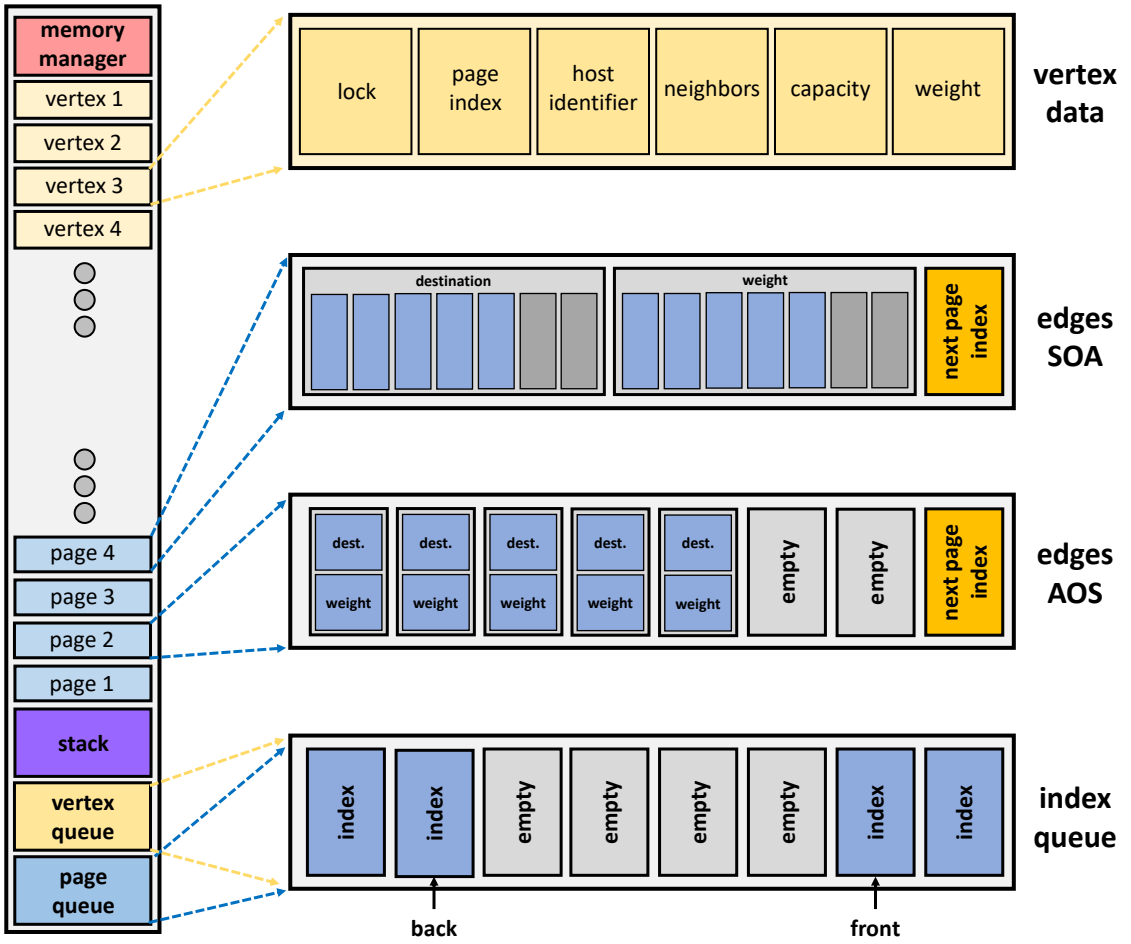


Figure 4.1: Visualization of the memory layout as employed by *faimGraph*. A *memory manager* is placed at the beginning of the manageable memory area, right after that, space is left for dynamic vertices. At the other end, two queues are instantiated for the purpose of page and vertex re-use as well as some configurable area usable as a stack for temporary data. Growing from the top down towards the vertices are the dynamic pages used to store adjacency data, which can either be stored in an *AOS* or *SOA* layout.

of variance over a large share of its vertices. Furthermore, both systems only consider adjacency data dynamic but leave the vertices static. But many modern use cases would greatly profit from the ability to consider the whole topology dynamic, being able to add or delete vertices in a graph as well as add new connections to them.

Considering these two limitations, we can evolve the initial design by introducing dynamic vertices as well as memory re-use for both edges and vertices alongside further improvements. This improved design is detailed in the following sections.

4.2 faimGraph

In the following section we discuss the design of *faimGraph* (**f**ully-dynamic, **a**utonomous, **i**ndependent **m**anagement of **graph**s). The focus is on the core contributions, which are a tight memory model building on the reuse of memory by utilizing *queueing* structures, dynamic changes of vertex and edge data, as well as high performance update implementations and algorithms running on top of the graph. An overview of the memory layout used by *faimGraph* can be see in Figure 4.1.

4.2.1 Memory Management

The central idea of *faimGraph* is performing all memory management directly on the GPU, requiring only a single allocation of a large block of memory to avoid round-trips to the CPU. This block serves all memory requirements for the graph structure itself as well as for algorithms running on top of the graph. During initialization, *faimGraph* prepares this memory, as shown in Figure 4.2, to support dynamic assignment and reassignment using *queueing* structures to keep track of unused memory. A *memory manager* is used to keep track of individual memory sections and current graph properties, like the number of vertices/edges and the currently largest vertex and page index in use in the system. Previously used, now free pages and vertex indices are tracked via queues, as discussed in Section 4.2.2. The majority of the memory is used by dynamically allocated *vertex data* and pages for *edge data*. Both regions grow from opposite sides of the memory region to not restrict the possible ratio between vertices and edges.

Temporary data (updates or helper data structures) can be placed in a *stack* which is preceded by two *queues* for reclaiming freed vertices and edge pages or directly after the vertex region if vertices are static for a procedure. If vertices are static, the stack region can even be omitted, lowering the overall memory requirements further. Since the complete addressing scheme uses relative indices, the framework can also be started with conservative memory bounds, as in most cases reinitialization is trivial. Reinitialization can be done directly from old *faimGraph* to new *faimGraph* building on just two `memcpy`'s on the device directly. If resources are even more scarce, reinitialization also can be performed (at a higher cost) from *device CSR*, *host CSR* or even *host* faimGraph. Reinitialization cost is discussed further in the evaluation, see Section 4.3.

4.2.2 Queues

The core entities for memory reclamation are the *index queues* used to store freed vertex indices and pages. Whenever a vertex is deleted or a page is freed, its index is pushed into the respective index queue. During resource allocation, threads at first attempt to pop a free element from the queue and only if that fails, increase the vertex or page region. Using this approach, changes in growth in the graph do not affect the required memory as

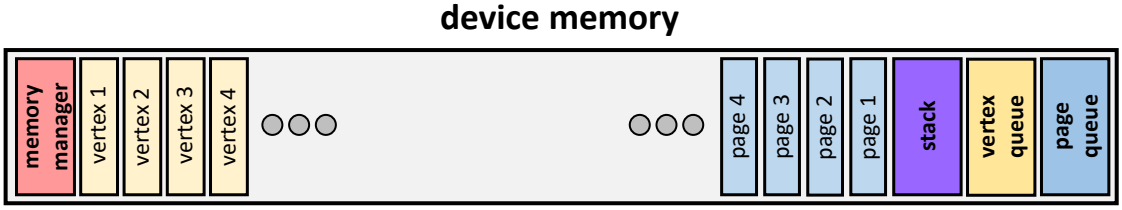


Figure 4.2: The manageable memory area consists of a memory manager unit at the beginning, followed by dynamic vertices and dynamic pages for edge data. At the end, a stack can be placed for temporary data and at the very end, two queues are located dealing with memory re-use for vertices and pages.

much as previous approaches would have, as a graph can grow in specific areas and shrink back in others. Furthermore, this allows for *O(1)* allocation of vertices as well as pages.

For efficiency, we use array-based queues, which operate on top of a ring buffer of indices, as can be seen in Figure 4.3. The queues must support concurrent access from thousands of threads and efficient queries for empty states. Thus, we use a front and back pointer as well as a fill counter for each queue, similar to the base queue design in the Broker queue [26]. Threads at first test the fill counter to determine whether there are elements in the queue. Only then, they atomically move the pointers to retrieve a queue element. As the entries in the queues are simple indices, we use *Atomic-Compare-And-Swap* (and *Atomic-Exchange* respectively) to insert or remove elements from the queue while using an empty flag to avoid read-before-write and write-before-read hazards.

4.2.3 Graph Data

4.2.3.1 Vertex Data

Other approaches, such as *aimGraph* [60] and *cuSTINGER* [22] consider vertex data as static. However, dynamic graphs may require the ability to add or remove vertices from the graph. Consider the example of load balancing in a communications network, where one might want to compute some metrics for a certain number of cell towers and all mobile

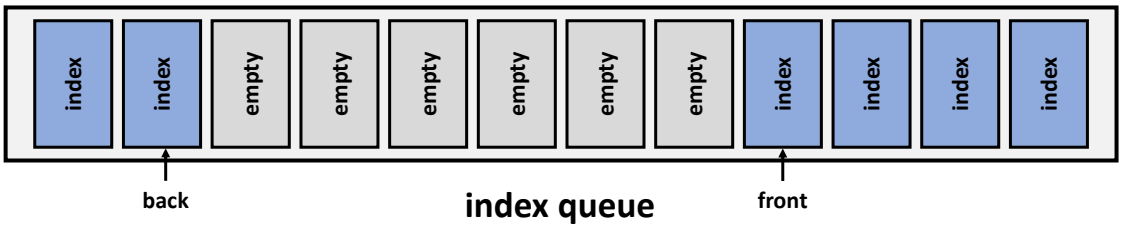


Figure 4.3: Both vertices and pages can be re-used by utilizing an index queue to keep track of the freed indices. This queue is array-based, built on top of a ring buffer using a front and back pointer (index) of static size.

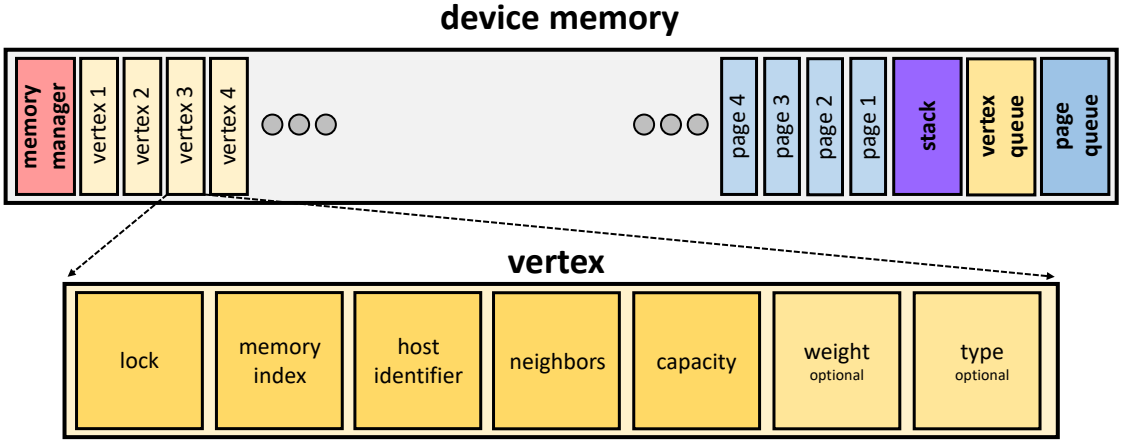


Figure 4.4: A vertex consists of a memory index (pointing to the first page holding adjacency data), a *host identifier*, neighbors and capacity detailing the current storage requirements as well as lock usable by algorithms. Depending on the graph type, it can also hold a weight and type.

devices connected to these towers. If a device moves out of range, we would like to remove this device from our consideration, vice versa, if another device moves into range, we would like to add it to the graph representation.

While a static vertex management can follow a *SOA* approach to enable efficient memory access to this data on the GPU, such an approach interferes with the concept of dynamic data distribution between vertex and edge data. One would have to choose a fixed array size to place these arrays one after the other in memory and increasing this size would entail a large overhead. Especially if memory is already scarce within the framework, the existing arrays might even have to be retired to host memory and then be copied back with separate copies into the enlarged regions. Thus, we store vertex data as a dynamically growing array-of-structures (*AOS*). Furthermore, individual structures in this array can be freed and reclaimed through the *vertex queue*, as detailed in Section 4.2.2.

Depending on the graph type, vertices may require different parameters, as can be seen in Figure 4.4. As the memory management is not bound to a specific vertex size, each vertex can hold as many parameters as the application requires. Allocation of new vertex indices can be achieved in *O(1)*. The procedure first queries the *vertex queue*, thereby reusing freed indices from previous deallocations. If the queue does not hold any available indices, we simply increase the dynamic array using an *Atomic-Add* on the vertex array size. Deleting a vertex includes deleting all edges referencing this vertex and returning its index to the *vertex queue* for later reuse. Depending on the *directedness* of the graph, this can be trivial in case of an undirected graph (as all references to the graph are implicitly known). But this can require significant overhead in case of a directed graph, as potentially every other vertex might reference the current vertex, in which case the deletion procedure is much more involved. Furthermore, all currently allocated pages to this vertex have to be

returned to the *page queue*, which incurs at least one traversal of the linked data structure per deleted vertex.

Keeping all vertex data next to another in memory has the advantage that simple indices can be used to reference vertices. The vertex's identifier used on the CPU is not bound to the memory location the vertex is stored at. We report a mapping between the host identifier and the device identifier back to the CPU after insertion. Additionally, when storing vertices sequentially, algorithms iterating over vertices show an efficient memory access pattern and better caching behavior.

4.2.3.2 Edge Data

Allocating individual vertices is reasonable as there is usually no direct commonality between different vertices and memory requirements can be kept as low as possible. This strategy makes less sense for edges as there are usually many edges originating from the same vertex, which will often be iterated sequentially and especially on a *SIMT* processor like the GPU, can be accessed much more efficiently next to one another in a coalesced manner. Thus, edge data is placed on *pages* of a fixed size and multiple pages form a linked list of edges for every vertex. The linked list nature allows for dynamic changes to the size of the adjacency.

This approach can be seen as a combination of a linked list and an adjacency array, yielding memory locality for edges within a page. At the same time, this strategy avoids reallocation of the whole adjacency if augmentation is required, by simply adding/removing a page to/from the linked list. As part of an efficient memory re-use scheme, a freed page is deallocated by enqueuing its index into the *page queue* for later re-use. The page size itself forms a trade-off between overallocation and efficiency. A smaller page allows for a tighter bound, closer to the actual number of edges per vertex, while a larger page size allows for more efficient traversal of the edges as the number of links to traverse is reduced. At the same time, a too small page size also increases the number of pointers to the next page (we use the last 4 B on each page). Thus, the most suitable page size is application dependent and can be chosen to fit different scenarios and will especially vary between different graph types.

For all our experiments, we chose a page size of 64 B. This one the one hand coincides with the memory alignment of *cuSTINGER* and provides a good balance between performance and overallocation per adjacency for simple graphs. On the other hand it is half of the *cacheline* size, so warp-based processing also should derive some benefit. For the adjacency data itself, we support two memory layouts, of which either may achieve better performance depending on the traversal characteristics of the graph algorithms. If multiple properties per edge are required, the *SOA* approach provides better memory access characteristics, as can be seen in Figure 4.6. On the other hand, as long as page size stays below or equal to the *cacheline* size, even an *AOS* layout will perform well, as can be seen in Figure 4.5. Also, vector-loads might me utilized to load in multiple properties,

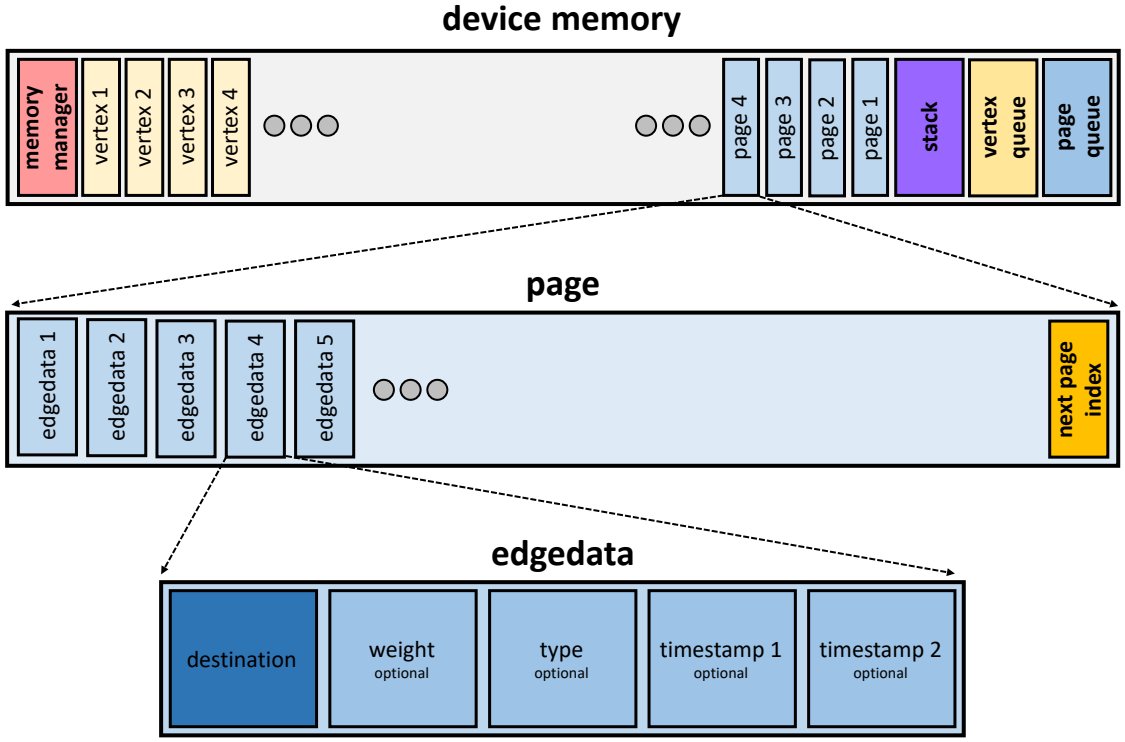


Figure 4.5: Edge data stored in *AOS* format on a page. Each edge consists off at least the destination, but can also hold, depending on the graph type, a weight, a type as well as two timestamps.

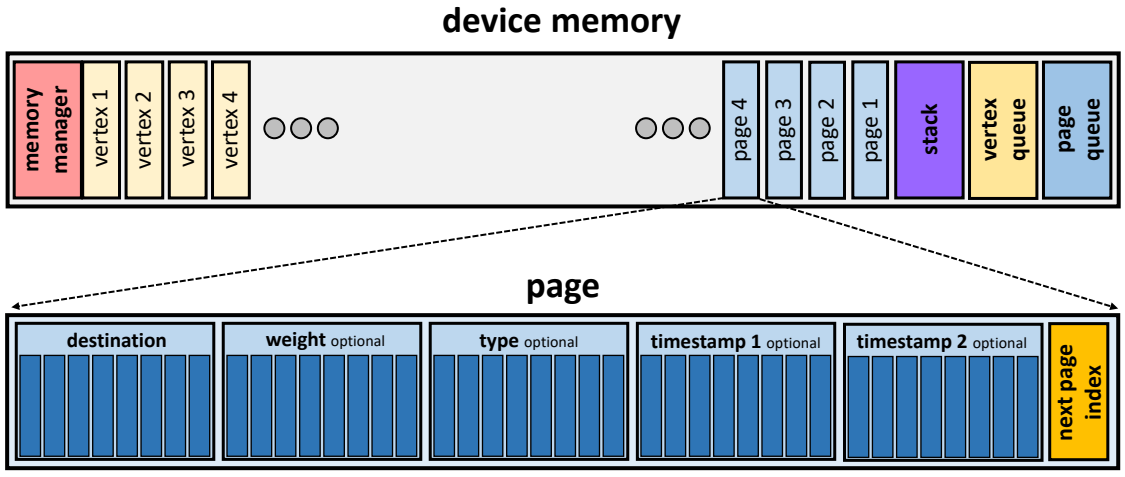


Figure 4.6: Edge data stored in *SOA* format per page, which can improve performance especially for semantic graphs due to a more efficient memory access pattern.

especially when considering a thread-based access model. Such properties include at least the destination vertex (simple graphs), weights (weighted graphs) and a type plus two timestamps (semantic graphs). For simple graphs, *AOS* and *SOA* are identical.

4.2.4 Vertex Updates

It is typical for graph structures to refer to vertices by their indices in memory, which alleviates look-up procedures to locate vertices. This increases the cost of updates as a mapping procedure is required that maps an arbitrary vertex identifier on the CPU to an index on the GPU. Moreover, deleting vertices also has to be reflected in the adjacency data by removing all entries referencing said vertices. While edges are organized as a linked list of pages that support locking, all vertices are organized in the same pool of memory and need to be updated in parallel to achieve high performance.

4.2.4.1 Vertex Insertion

Vertex insertion is based on a four step approach to achieve parallel insertion, starting by sorting the update data batch (line 3), as can be seen in Algorithm 3. The next two steps are concerned with duplicate checking, while the fourth performs the insertion itself. Duplicates can occur within a batch of to-be-inserted vertices (line 4) and with vertices already present in the graph (line 5). Duplicates with the graph are non-trivial to find due to the mapping between CPU and GPU vertex identifiers. It would be very inefficient to search the entire GPU vertex structure for each to-be-inserted vertex.

Thus, we propose a *reversed* duplicate check with the graph vertices. Given that the batch of to-be-inserted vertices is already sorted, searching in the batch is rather efficient. Thus, we start one thread for each graph vertex, which looks up its mapping from GPU to CPU identifier and performs a binary search on the sorted to-be-inserted vertices. If a duplicate is found, it is simply marked in a helper data structure to not hinder the

Algorithm 3: Vertex Insertion

```
   Data: Vertex Update Batch
   Result: Vertices inserted into graph
1  Copy Vertex updates onto stack;
2  if sorting_enabled then
3      thrust::sort(vertex_updates);
4  d_duplicateCheckingInBatch (vertex_updates);
5  d_reverseDuplicateCheckingInGraph (vertex_updates, graph);
6  d_vertexInsertion (vertex_updates, graph);
7  Copy mapping back to host;
8  if sorting_enabled then
9      Copy vertex updates back to host;
```

subsequent checking step. Next, duplicate checking within the batch is performed and one thread for each entry is started. Each thread checks its batch successor and if a duplicate is found, simply marks it as a duplicate directly in the batch and continues as long as it finds duplicates in successive order. As the batch is sorted, this leaves only the first element of multiple duplicates remaining. After both steps, the helper data structure is synchronized with the vertex update batch, removing duplicates with the graph from the batch as well. The actual vertex insertion process (line 6) is straightforward: The framework starts by acquiring a new device index and a new page index for each valid vertex update.

Both first contact the respective queues for previously deleted indices. If a queue is empty, the memory manager supplies fresh indices from the back of the currently allocated state. Then, the vertex is set up using the update data and the adjacency page is inserted. Finally, the new mapping from host identifier to device identifier, i.e., each vertex's position in the vertex array, is reported back to the host.

4.2.4.2 Vertex Deletion

As each deletion procedure performs an *Atomic-Compare-And-Swap* on the host identifier, only one thread will retrieve a valid identifier and continue the procedure, alleviating the need for duplicate checking. In contrast to vertex insertion, deleting a vertex not only alters vertex management data, but also has implications on adjacencies. This results from the fact that other vertices can reference said vertex by possessing an edge to it. These references have to be deleted from the graph as well and the holes left by these deletions have to be compacted in a separate procedure, as can be seen in Algorithm 4.

In case of an *undirected* graph, these references can be deleted directly in the deletion procedure (line 4), as each adjacency element has a dual that can directly be found by simply swapping the source and destination of an edge. The procedure iterates over the adjacency of the to-be-deleted vertex and for each edge it removes the dual in the corresponding adjacencies. As no duplicates are present in the adjacencies, this deletion

Algorithm 4: Vertex Deletion

```
   Data: Vertex Update Batch
   Result: Vertices deleted from graph
1  Copy Vertex updates onto stack;
2  if sorting_enabled then
3  |  thrust::sort(vertex_updates);
4  d_vertexDeletion (vertex_updates, graph);
5  if graph_is_directed then
6  |  d_reverseDeleteVertexMentions (vertex_updates, graph);
7  d_compaction (graph);
8  Copy mapping back to host;
```

can even be performed without locking. If, on the other hand, the graph is *directed*, the deletion procedure is not as straightforward and we use a multi step approach (line 6).

For a directed graph, arbitrary vertices may reference a to-be-deleted vertex. Thus, in a first step, we only return the pages allocated for the vertex to the page queue. The update data batch is once again sorted to speed up the following step (line 3). Once again, we propose a *reversed* deletion process similar to the *reversed* duplicate check, starting a worker per adjacency and searching each edge in the sorted update batch, which is once again rather efficient.

After the actual deletion, the framework still has to perform compaction on the adjacencies (line 7). To avoid unnecessary locking during this step, the actual clean up is performed in a separate kernel by iteratively moving edges from the back to empty positions in the adjacency (or moving edges to the front consecutively to respect sort order). Again, using more than a single thread for this operation can increase performance. Finally, the now free vertex index is returned to the *vertex queue* and the mapping change is reported back to the host.

4.2.5 Edge Updates

Edge updates are considered a common operation for dynamic graphs. In *faimGraph*, update information is considered to be made available to the framework by either the CPU-side and successively copied to the GPU or directly in a GPU buffer. The update procedure runs independently on the GPU in either case. A benefit of this methodology, in addition to alleviating additional management interventions from the host, is the fact that users do not need to care about memory management. Similarly to *aimGraph*, vertex structures hold a lock and update threads can lock each adjacency list before altering it to gain exclusive access. However, *faimGraph* adds support for multiple coordinated threads to alter adjacencies, which is preferable when scanning larger adjacency sizes. As coordinated threads need to communicate, we either use cooperative thread blocks or warps (groups of threads executing on the same *SIMD* unit). We call this kind of strategy an *update-centric* approach, as each update is mapped to an individual *worker* (thread/warp/block). Locking strategies work well if

- the update pressure is not particularly high (updates are distributed over the graph well)
- the average size of the adjacencies is rather small (less than ≈ 25 according to our experiments)
- and the graph is well balanced

If updates in a batch favor a smaller set of vertices, the overhead introduced by locking as well as multiple adjacency traversals becomes a serious bottleneck.

Thus, we propose a new update strategy that avoids locking by coordinating the update efforts beforehand: the *vertex-centric* approach. It devises an offset scheme to start a worker per vertex that is affected by updates. In this way, exclusive access to each adjacency is guaranteed and waiting on locks is avoided. Different update implementations can also be mixed for consecutive update calls to deal with changing requirements.

4.2.5.1 Edge Insertion

Our *vertex-centric* edge insertion splits the insertion process into three steps. First, an offset scheme is constructed: We sort the insertion requests according to the source vertex in-place. A following prefix sum determines the offset of each specific source vertex in the sorted array and the number of updates that will be performed for a specific adjacency. Second, duplicate checking is performed, which—if activated—makes sure that edges are only added to the graph once. To this end, the insertion requests are compared to the already existing graph and to the other requests in the sorted array. Third, one worker per affected vertex is started, which adds the edges to the end of the adjacency lists. If there is still sufficient space on the last page of the adjacency list, the edges are added to this page, otherwise additional pages are queried first from the *page queue*. If no freed pages are currently available, a new page index is supplied by the memory manager. This significantly reduces the update time and inserting millions of edges can be performed in a matter of milliseconds.

Furthermore, building on this approach it is also possible to respect sort order when inserting new vertices. During insertion, as can be seen in Figure 4.7, a combined sweep over the to-be-inserted and already present edges is then sufficient to insert the data: If an edge must be inserted before the end of the adjacency list, we simply swap the edge currently in this slot with the update edge and merge the replaced edge into the insertion requests, hence the sorting effort is constrained to the update data targeting this specific vertex. Thus, when the end of the adjacency is reached, the remaining to-be-inserted edges can be placed at the back. Note that a sorted adjacency does not require separate duplicate checking, as the entire existing adjacency is scanned during insertion anyway.

Nevertheless, the determining factor for performance remains update pressure and adjacency traversal. High update pressure and longer traversal lend themselves to the *vertex-centric* approach, while, otherwise, *update-centric* provides better performance.

4.2.5.2 Edge Deletion

Vertex-centric edge deletion starts with the same sorting and prefix sum steps as *vertex-centric* edge insertion. Duplicate removal is not necessary for edge deletion, as edges can only be removed once. When a to-be-removed edge is found, we simply copy the last edge from the adjacency list over the edge to avoid holes in the list. If, on the other hand, we want to respect sort order, we instead iteratively shuffle all remaining edges to the front, overwriting the to-be-deleted elements in the process. Again, the entire operation is

performed in a single sweep over the edge data. If pages are left empty after the compaction step, they are returned to the *page queue*.

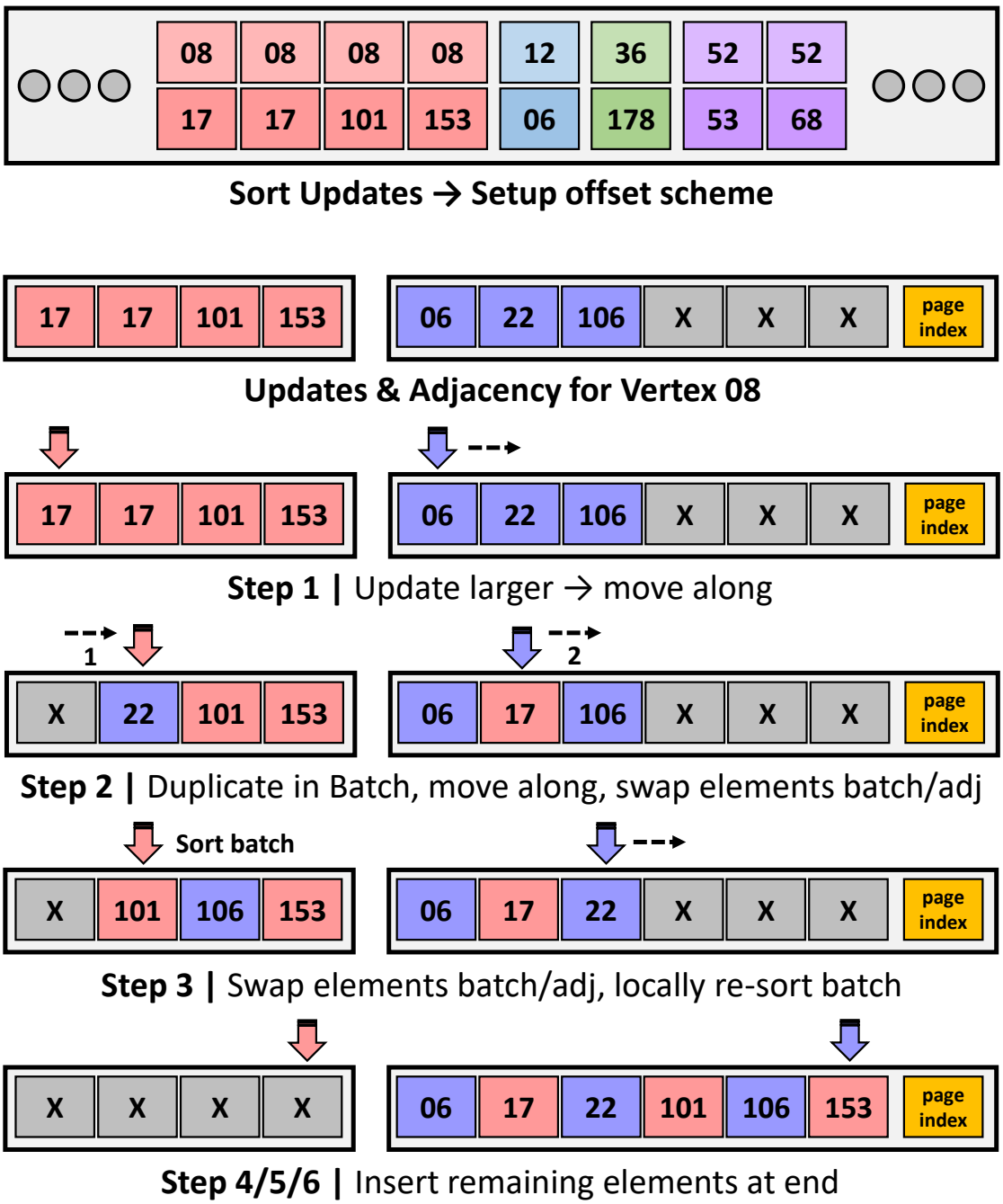


Figure 4.7: Example for sorted insertion with duplicates in batch using the *vertex centric* approach.

	hugetr.	*germany*	*luxemb.*	*europe*	*del*_n23*	*del*_20*	*coAuth.*
Type	Simul.	Road	Road	Road	Triang.	Triang.	Citation
—V—	5.82M	12M	115k	50.91M	8.38M	1.04M	227k
—E—	8.73M	24.74M	239k	108.1M	25.16M	3.14M	815k
avg. deg(V)	1.50	2.06	2.08	2.12	3.00	3.02	3.59
faimGraph							
Initialization (ms)	**20.21**	**32.78**	**1.16**	**145.16**	**45.88**	**6.38**	**2.47**
Initialization (MB)	466.59	925.16	9.18	4078.4	672.01	84.0	**20.41**
Reinit. 105% (ms)	5.18	9.42	0.36	26.75	9.21	1.41	0.58
Uniform (MB)	**467.16**	925.17	**41.97**	4078.4	675.01	103.45	**49.97**
Random (MB)	**467.74**	925.18	**50.73**	4078.4	**676.98**	**114.98**	**60.06**
Sweep (Rounds)	**all**	**all**	**all**	**all**	**all**	**all**	**all**
Queries (ms)	3.28	3.23	2.61	3.21	3.23	3.19	3.26
V. Insertion (ms)	6.75	11.33	1.99	45.68	9.42	2.85	2.09
V. Del. (UD) (ms)	7.07	11.54	1.49	43.94	10.42	2.90	2.29
V. Del. (D) (ms)	8.67	15.75	1.26	69.17	18.95	2.89	2.36
aimGraph							
Uniform (MB)	467.83	925.18	49.13	4078.4	678.40	122.17	59.82
Random (MB)	468.22	925.19	61.14	4078.4	678.88	136.88	72.35
Sweep (Rounds)	2481	2367	**all**	1523	2429	2584	**all**
cuSTINGER							
Initialization (ms)	6177.8	12752	94.48	64124	9286.9	982.69	191.32
Initialization (MB)	844.32	1674.1	16.61	7380.1	1216.1	152.03	36.81
Uniform (MB)	855.76	1675.6	66.86	7380.1	1264.2	216.32	87.97
Random (MB)	847.19	1674.2	69.52	7380.1	1236.7	208.44	89.11
Sweep (Rounds)	1898	1732	**all**	593	1824	2049	2060
GPMA							
Initialization (ms)	121.14	181.23	34.27	782.85	308.22	50.61	55.24
Initialization (MB)	**232.92**	**372.78**	**4.68**	**1634.84**	**562.04**	**70.25**	26.92
Uniform (MB)	—	**400.94**	209.79	**1650.84**	**587.97**	**90.38**	106.16
Random (MB)	—	**621.70**	282.73	**1915.54**	843.84	294.00	299.92
Sweep (Rounds)	28	49	176	38	136	264	73
Hornet							
Initialization (ms)	300.87	434.94	4.20	4410.6	468.20	45.99	10.73
Sweep (Rounds)	5862	1641	**all**	1447	3314	3998	**all**

Table 4.1: Performance measurements for *cuSTINGER*, *Hornet*, *GPMA*, *aimGraph* and *faimGraph*, including initialization time and overall timings for a complete test set as well as memory evaluation on three test cases on the graphs *hugetric-00000*, *germany*, *luxembourg*, *europe*, *delaunay_n23*, *delaunay_20*, *coAuthorsCiteseer*

	coAuth.	*rgg_n.*	*nlpkkt200*	*nlpkkt120*	*nlpkkt240*	*ldoor*	*audikw1*
Type	Citation	Geom.	Matrix	Matrix	Matrix	Matrix	Matrix
—V—	299k	1.04M	16.24M	3.5M	27.99M	952k	943k
—E—	1.95M	6.89M	431.9M	93.3M	746.4M	45.57M	76.71M
avg. deg(V)	6.52	6.63	26.59	26.66	26.67	47.87	81.35
faimGraph							
Initialization (ms)	**2.61**	**11.60**	**459.36**	**100.03**	**792.88**	**58.11**	**102.92**
Initialization (MB)	**26.44**	**99.98**	**2277.3**	**494.62**	**3929.8**	**244.66**	**355.53**
Reinit. 105% (ms)	0.613	2.38	27.23	10.49	36.04	6.10	8.90
Uniform (MB)	**54.44**	**121.97**	**2349.1**	**544.83**	**3980.7**	**235.30**	**359.99**
Random (MB)	**65.03**	**138.13**	**2372.6**	**569.59**	**3996.3**	**257.22**	**382.80**
Sweep (Rounds)	**all**	**all**	**all**	**all**	**all**	**all**	**all**
Queries (ms)	3.12	3.21	3.23	3.28	3.24	4.58	5.73
V. Insertion (ms)	2.15	2.87	15.19	4.87	25.63	2.82	2.80
V. Del. (UD) (ms)	2.82	4.01	52.98	22.31	74.27	12.31	29.63
V. Del. (D) (ms)	2.02	4.03	83.15	20.84	144.15	11.87	24.54
aimGraph							
Uniform (MB)	65.68	149.99	2430.5	608.64	4038.4	267.66	392.69
Random (MB)	78.11	163.31	2410.1	625.33	4014.8	284.52	411.06
Sweep (Rounds)	**all**	2580	2033	2473	1589	2547	2518
cuSTINGER							
Initialization (ms)	254.37	944.55	20907	3848.9	—	966.58	1013.4
Initialization (MB)	47.71	195.34	4292.8	930.19	—	369.99	548.99
Uniform (MB)	100.02	247.44	4295.2	948.15	—	381.70	553.93
Random (MB)	99.78	245.41	4293.6	936.91	—	378.38	551.27
Sweep (Rounds)	2057	2046	1262	1892	—	1999	1951
GPMA							
Initialization (ms)	52.92	86.08	—	467.94	—	224.81	387.50
Initialization (MB)	30.87	137.68	—	885.78	—	422.51	702.64
Uniform (MB)	105.84	157.33	—	996.23	—	—	731.64
Random (MB)	257.89	286.75	—	1133.00	—	—	1110.37
Sweep (Rounds)	68	29	—	60	—	11	19
Hornet							
Initialization (ms)	13.18	61.72	53k	755.97	230k	168.59	207.06
Sweep (Rounds)	**all**	2899	4715	5701	3955	6504	3140

Table 4.2: Performance measurements for *cuSTINGER*, *Hornet*, *GPMA*, *aimGraph* and *faimGraph*, including initialization time and overall timings for a complete test set as well as memory evaluation on three test cases on the graphs *coAuthorsDBLP*, *rgg_n_2_20_s0*, *nlpkkt200*, *nlpkkt120*, *nlpkkt240*, *ldoor* and *audikw1*

4.3 Evaluation

In this section, we evaluate the performance of *faimGraph* and compare it to *aimGraph* and the publicly available *cuSTINGER*, as well as to *Hornet* and *GPMA* wherever possible. The performance measurements were conducted on an NVIDIA Titan Xp (12 GB V-RAM), and an Intel Core™ i7 -7770. The graphs used are listed in Table 4.1 and Table 4.2. They represent a cross section of different problem domains and were taken from the *10th DIMACS Graph Implementation Challenge* [5].

4.3.1 Memory footprint

One of the biggest differences between *faimGraph* and previous approaches is memory consumption and memory footprint over time, as can be seen in Figure 4.8. Although *faimGraph* starts with a larger allocation as it manages memory directly on the device, the actual memory footprint within the framework (especially over time) is lower compared to previous approaches and all memory allocations are facilitated directly through the framework without host intervention. Since the cost of reinitialization is negligible in most cases (due to the relative addressing within the pool which allows the usage of just two `memcpy`'s to reinitialize), even the initial allocation can be chosen conservatively. *cuSTINGER* performs sequential allocation calls from the CPU to allocate the management data and all individual edge blocks in the initialization procedure. Especially for graphs

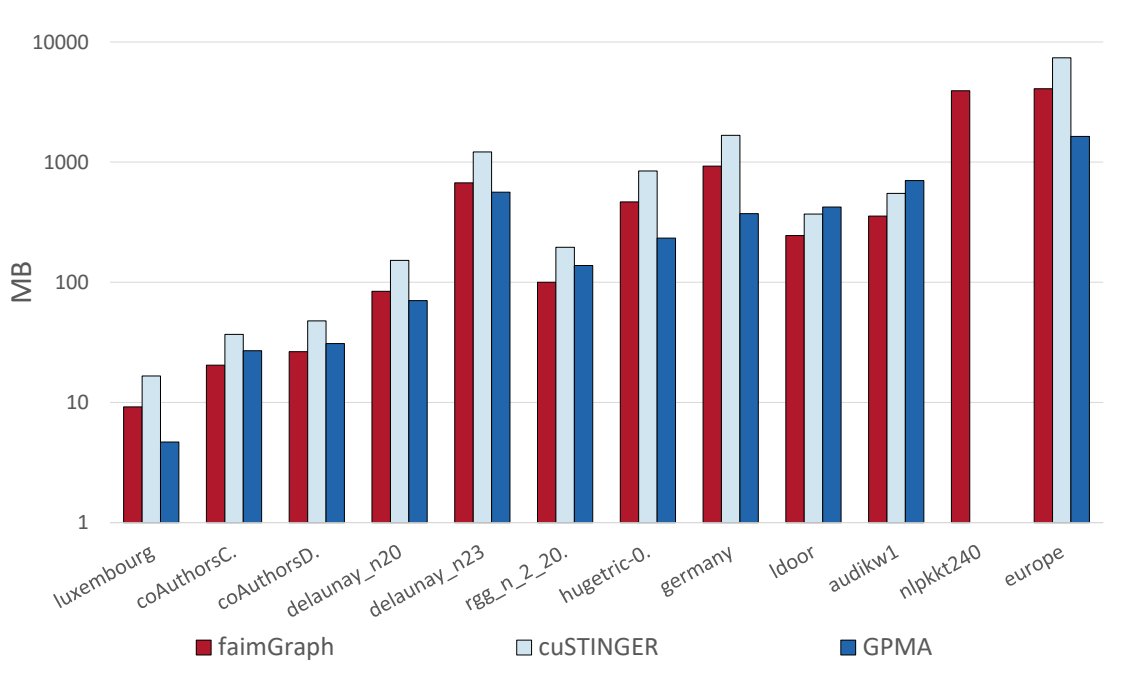


Figure 4.8: Memory footprint after initialization

with millions of vertices, this is a significant overhead, compared to the single allocation in *faimGraph*.

Furthermore, due to the overhead associated with reallocation, *cuSTINGER* uses overallocation to reduce the run-time cost for edge updates. *faimGraph* locates all its data by combining an efficient indexing scheme and reinterpreting memory on the fly. This way, the same functionality can be achieved with significantly less memory. Table 4.1 and Table 4.2 note the respective memory footprints within the framework for *faimGraph*, *cuSTINGER* as well as *GPMA*. The difference is most significant for high numbers of vertices (e.g.,, `europe` (14) with 4GB vs 7GB) for *cuSTINGER*, but also for large adjacencies due to overallocation (e.g.,, `nlpkkt200` (12) with 2 GB vs 4 GB for *cuSTINGER* or `audikw1` with 250 MB vs 420 MB for *GPMA*) for both *cuSTINGER* and *GPMA*. *GPMA* performs well for very sparse graphs as it stores no additional vertex properties (e.g., number of edges per adjacency), but experiences significant overhead for denser graphs as each edge has to store both source and destination as part of the *PMA*.

4.3.2 Memory usage evaluation

faimGraph's memory management scheme allows for reuse of memory over time. This is especially crucial for long-term use cases, where certain areas of the graph grow and shrink significantly. Both *aimGraph* and *cuSTINGER* hold the maximal allocation state in memory, meaning that once allocated, memory stays with its vertex. Hence, after prolonged usage the allocated memory resources do not reflect the actual memory requirements and may even lead to system failure over time.

To test long term use, we use three different test cases: The *Uniform* test case performs successive edge insertions and deletions derived from a uniform distribution. *Random* performs the same operations, whereas each round is randomly chosen to be either insertion or deletion. The memory footprint for these tests is shown in Table 4.1 and Table 4.2. *aimGraph* is more efficient in all cases compared to *cuSTINGER*, requiring between 12% to 45% less memory. *faimGraph* reduces the memory consumption further to 27% to 52% less memory compared to *cuSTINGER*. The *Sweep* testcase highlights the behavior for strongly volatile graphs. Each update round targets a set of 100 vertices with a batchsize of 1.000.000, where a set of edges is first inserted and then deleted again. Each update targets a successive set of source vertices. Performance is measured in rounds (how long can this procedure be repeated before the system goes out of memory). As shown in Table 4.1 and Table 4.2, *faimGraph* can run to completion for all graphs as the memory footprint after each round is mostly equal to the initial state. *aimGraph* and *cuSTINGER* fall significantly behind and only manage to complete all rounds within the 12 GB of memory for graphs that only require less than 100 MB by itself—the memory that *faimGraph* returns to after every deletion step during the sweep test. For these small graphs, our tests thus revealed a memory increase of more than two orders of magnitude above the necessary. This clearly underlines that fully dynamic memory management is essential

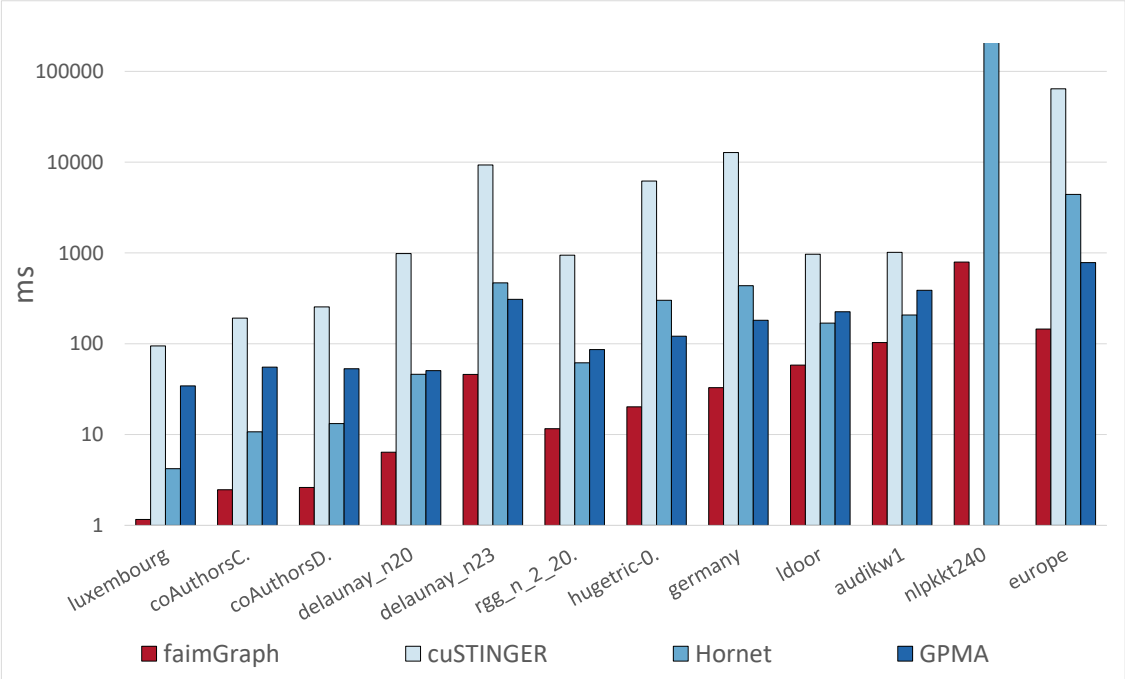


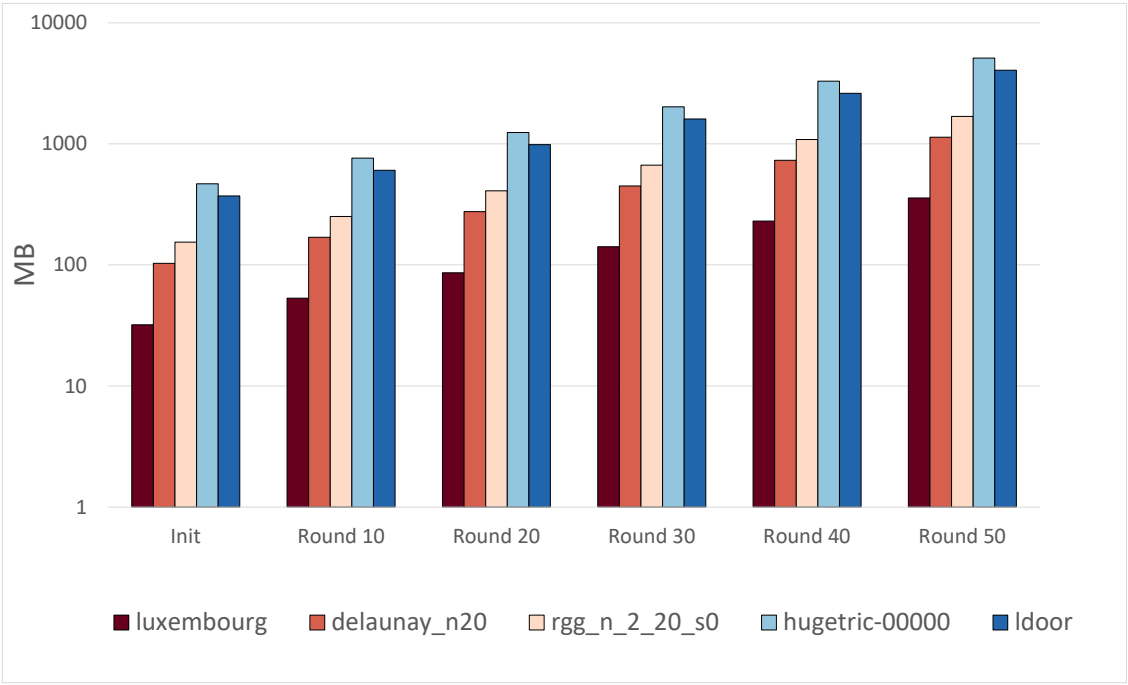
Figure 4.9: Time required to initialize a graph on the GPU.

in highly volatile problem domains.

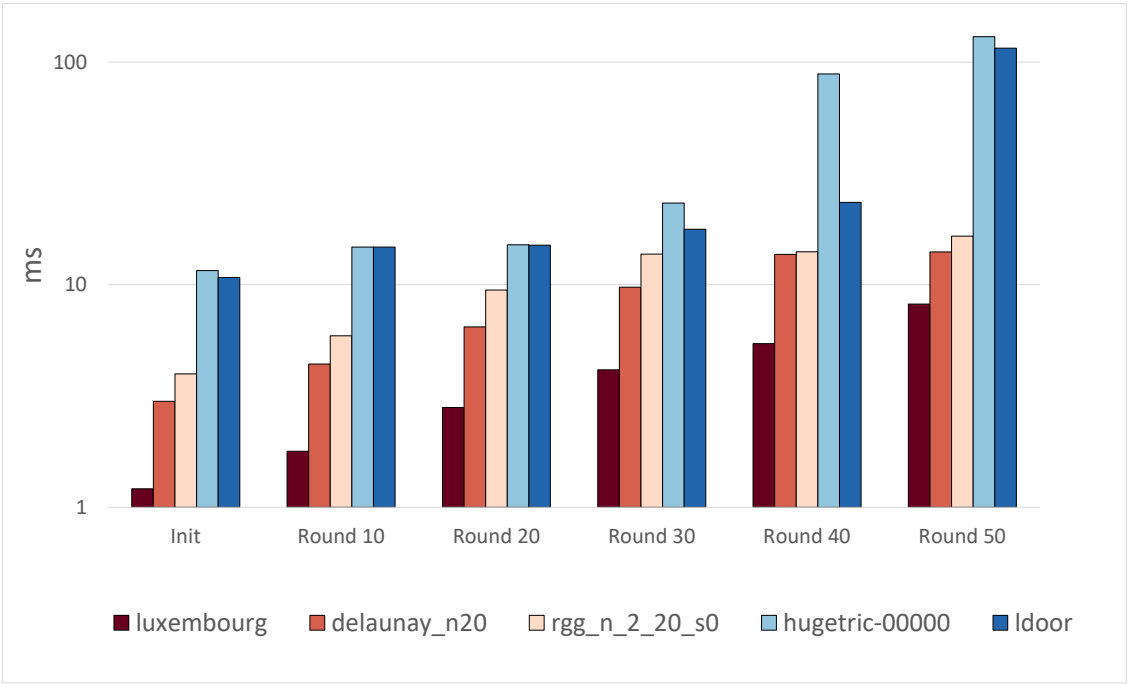
Using a large semi-continuous array, *GPMA* should able to re-use freed memory in the *Sweep* testcase as well. Unfortunately, since memory is localized, significant rebalancing might be required and subsequently, performance would be penalized. The current implementation does not handle duplicates within the update batch, but even correcting for that unfortunately still accumulates memory and we were not able to trace the source of this issue. At the time of this evaluation, there was only limited information available about *Hornet*, hence no actual memory footprint could be determined. But looking at the *Sweep* testcase, it is clear that it provides an improvement over its predecessor *cuSTINGER*.

4.3.3 Initialization

The autonomous approach to memory management on the GPU pays off during initialization. *faimGraph* distributes memory to individual vertices fully in parallel and the single GPU memory allocation drastically reduces allocation overhead. *cuSTINGER* performs sequential iterations over all vertices to allocate its adjacencies. In all tested scenarios (*cf.* Table 4.1, Table 4.2 and Figure 4.9), *faimGraph* is able to outperform all other approaches by a significant margin. The same is true for reinitialization with increased size, which is even faster than pure initialization due to our favorable relative indexing setup. The discrepancy in performance (up to two orders of magnitude) is greatest for graphs with a large number of vertices, like `germany` and `europe`.



(a) Memory footprint after repeated reinitialization with 105 %.



(b) Performance overhead of repeated reinitialization with 105 %.

Figure 4.10: Overhead introduced after repeated reinitialization with 105 % of the base size for a selection of graphs.

But still in small dense graphs, e.g., `ldoor`, the speed-up is one order of magnitude. Performance overhead for reinitialization with 105% of the conservative allocation size is displayed in Table 4.1 and Table 4.2 as well as in Figure 4.10a and Figure 4.10b.

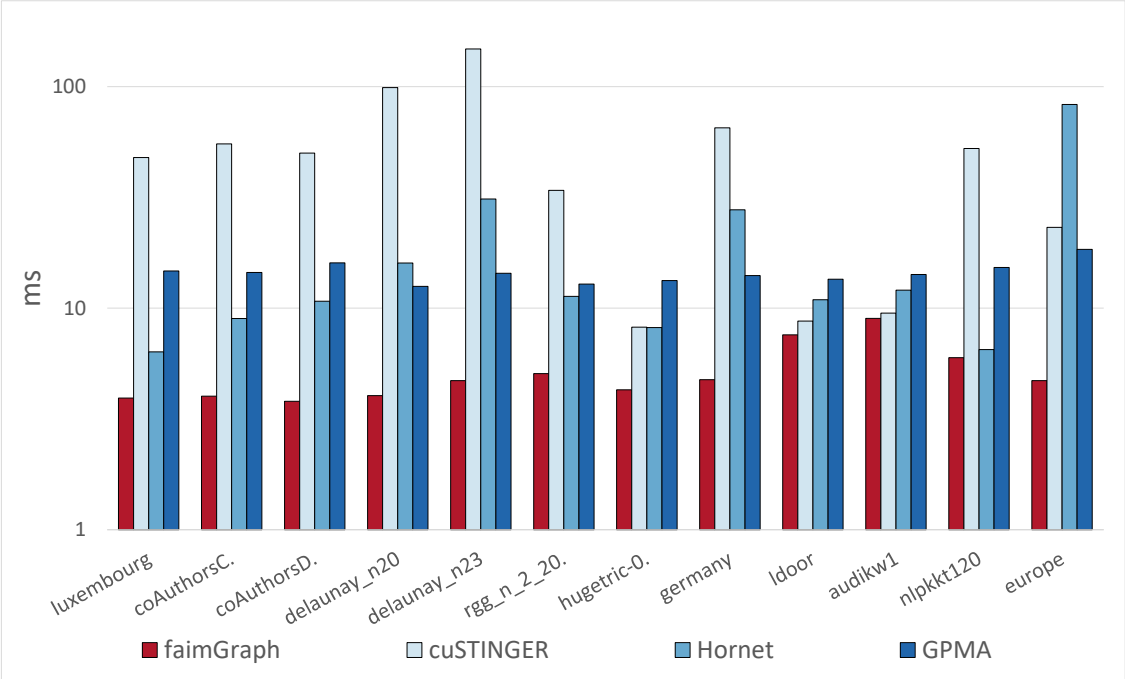
4.3.4 Edge Updates

Figure 4.11 and Figure 4.13 show the insertion and deletion performance for *faimGraph* as well as *cuSTINGER*, *GPMA* and *Hornet* for uniform and focused update distributions. *faimGraph* utilizes the conservative memory allocation with 50% additional pages for the initial allocation in these tests, due to efficient memory re-use none of the cases experienced reinitialization.

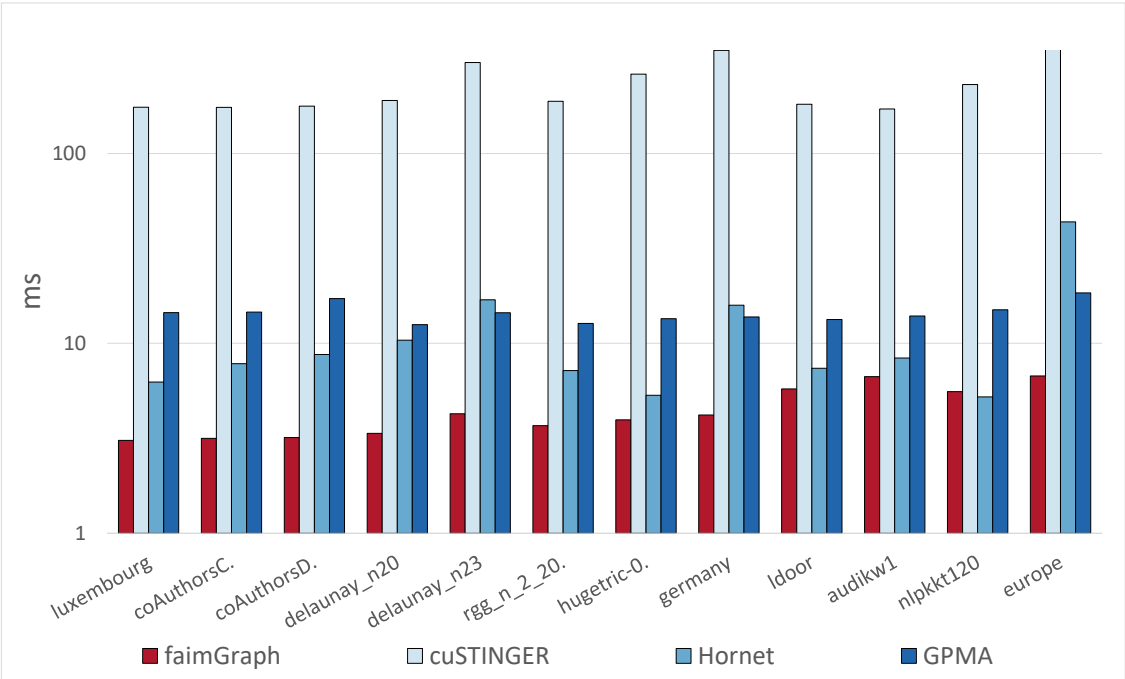
4.3.4.1 Edge Insertion

The clearest performance difference for edge insertion can be observed for graphs with small to medium sized adjacencies, which can be attributed to two factors: *cuSTINGER* performs an additional indirection step to follow the data structure and employs overallocation to reduce the need for reallocation. For larger graphs, the probability for insertions to hit the same vertex is smaller and thus, reallocation does not happen at all for *cuSTINGER*. For smaller graphs, on the other hand, the performance numbers quite clearly reflect the overhead introduced by reallocation procedures. *faimGraph* on the other hand directly interprets memory as required and does *not* employ overallocation. The *vertex-centric* approach is not particularly well suited to uniform updates with low update pressure. This results from the excessive duplicate checking needed in this case. The *sorted* approach performs exceptionally well, especially for small to medium sized adjacencies, as the benefits derived from sorted adjacencies and updates during the procedure outweigh the re-sorting effort. Thus, keeping sorted adjacencies actually introduces no to hardly any overhead. Only for large adjacencies the performance scales negatively with the increased memory access needed by the sorting procedure.

Increasing the update pressure (focusing the updates on a range of 1000 vertices, which sweeps over the graph throughout the test), as shown in Figure 4.11b, yields consistently good results for *faimGraph*, even with the *update-centric* approach. Although the locking overhead is clearly visible in all cases, locking still outperforms *cuSTINGER* in all cases as the reallocation procedures are handled more efficiently. Both *vertex-centric* approaches outperform the *update-centric* approach due to reduced overhead while traversing the adjacency and the removal of locking. *faimGraph* outperforms *cuSTINGER* by a factor of 1.1–114× / 1.1–31× for both batch sizes with uniform edge insertion; focussing updates on a smaller range of vertices yields a speed-up between 25–185×. Similarly, the speed-up of uniform updates compared to *GPMA* is 2.5–14× / 1.6–4.2×; with higher update pressure the difference is 2.1–5.4×. Compared to *Hornet*, the speed-up achieved for uniform updates is 1.6–16.5× / 1.1–17.6×, using higher update pressure updates *Hornet* performs slightly better in one case, the overall speed-up falls between 0.93–6.48×. In summary, it can be



(a) Edge insertion performance with random vertex source.



(b) Edge insertion performance with vertex source focused on a small range (1000) of source vertices for higher update pressure.

Figure 4.11: Edge Insertion with a batch size of 1 000 000

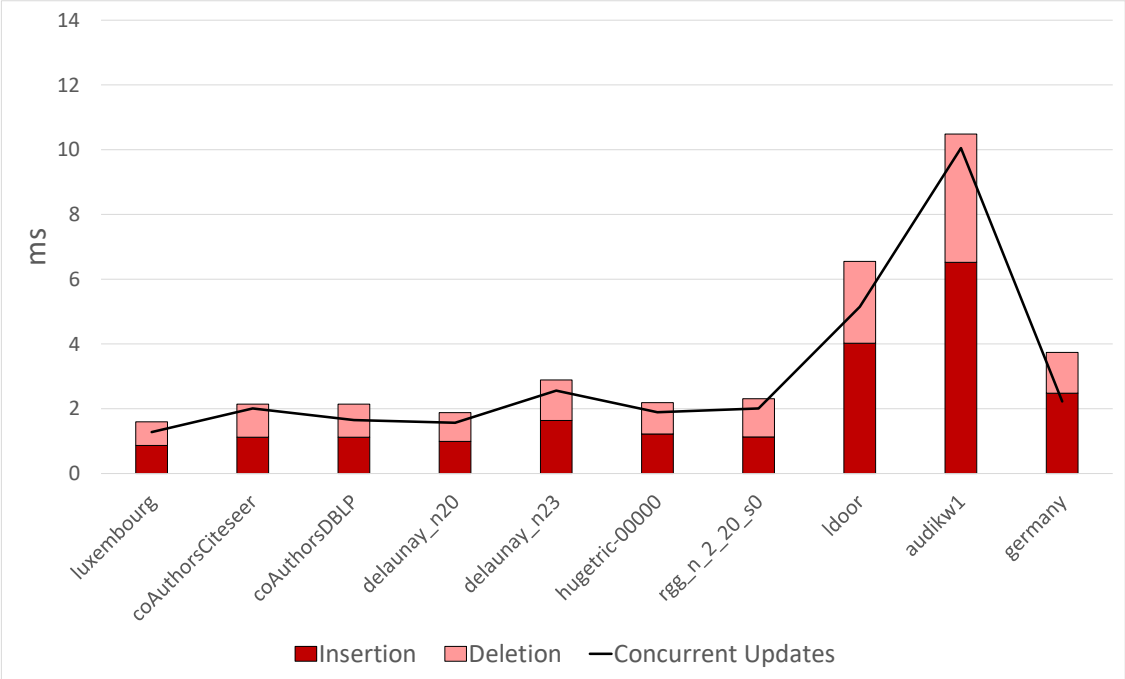
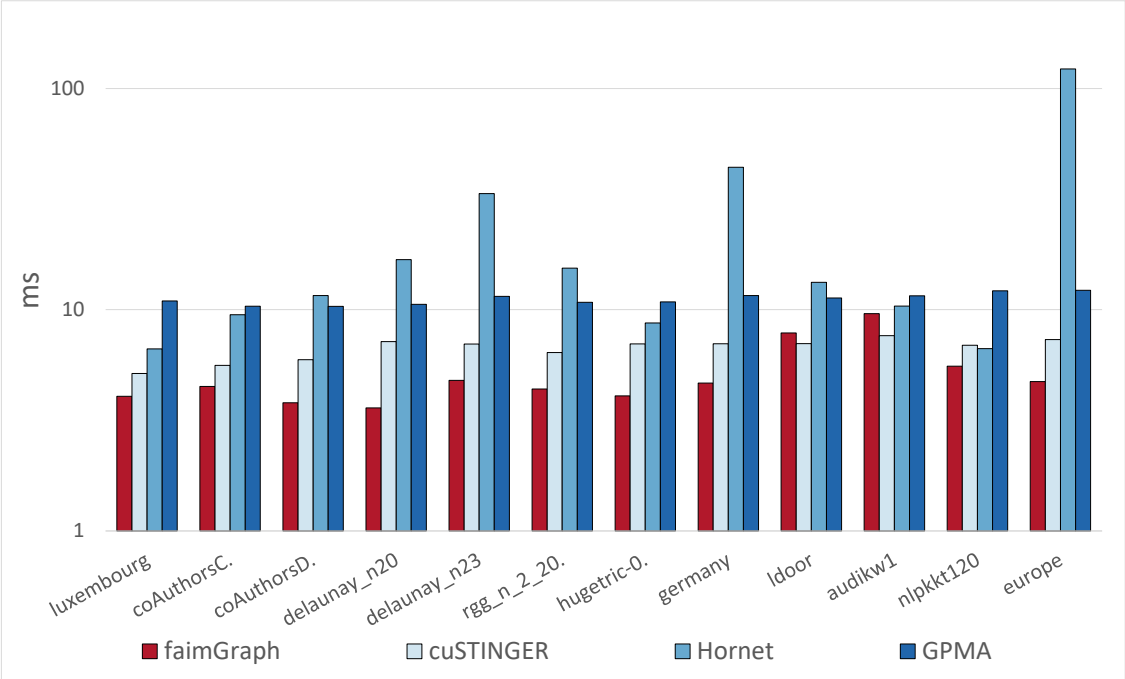


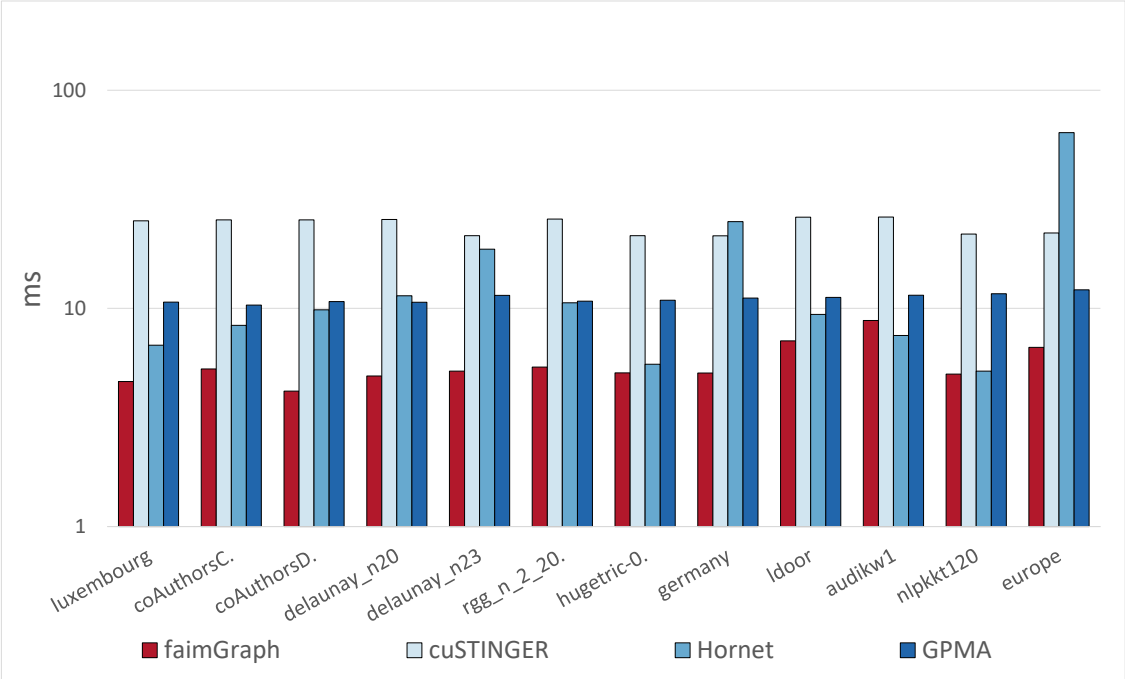
Figure 4.12: Concurrent edge insertions and deletions for a batch size of 100 000.

noted that *cuSTINGER* performs well when it works within its overallocation boundaries, but otherwise drops significantly. *GPMA* performance is very uniform independent of the sparsity of the graph, but is slower overall. *Hornet* performs well if the source vertex range modified is small, but falls behind significantly for large source vertex ranges in sparse graphs. The best *faimGraph* strategy is always faster than *cuSTINGER*, which can easily be selected based on the update pressure. Higher update pressure clearly favors our new *vertex-centric* approaches.

Lastly, we also tested concurrent edge insertions and deletions within *faimGraph*, as can be seen in Figure 4.12. We compare performing edge insertion and deletion after one another compared to performing them simultaneously. This reduces the overhead of separate kernel launches but also keeps the memory requirements lower, as deletions might free up space that can then be occupied by insertions later on. Overall we can see that performance improves compared to performing edge updates separately. This mode might be improved even further by some pre-processing but is also in need of additional semantics (it might happen that an edge is inserted and in the next call removed again, in the concurrent scenarios the adjacency might or might not have the edge, depending on the order of the operations).



(a) Edge Deletion focusing on the whole graph.



(b) Edge Deletion focusing on a smaller range of source vertices, simulating higher update pressure.

Figure 4.13: Edge Deletion with a batch size of 1 000 000.

4.3.4.2 Edge Deletion

In case of deletions, the performance difference is slightly less pronounced and much more consistent compared to the insertion process. This is not surprising as edge deletion is very straight forward in *cuSTINGER*, as memory is not freed and duplicate checking is not necessary. *faimGraph* on the other hand additionally performs compaction and frees not needed pages, which introduces additional overhead. However, due to the smaller memory footprint and more efficient implementation, *faimGraph* still outperforms *cuSTINGER* in 10 of the 12 test cases for uniform deletion (Figure 4.13a) with a performance difference between 0.92× - 1.4× / 0.8× - 1.9×. Compared to *GPMA*, *faimGraph* is always faster with a speed-up of 1.9× - 3.6× / 1.4× - 2.9×. The same is true for *Hornet*, the difference here is 1.7× - 25× / 1.1× - 25.9×. Sorting again hardly reduces performance compared to the *vertex-centric* approach unless large adjacencies need to be moved as in `ldoor` (4) and `audikw1` (5). For larger graphs, with few updates per vertex, the locking strategy performs best. For high update pressure (Figure 4.13b), both *vertex-centric* approaches perform best in all test cases, clearly outperforming both the *update-centric* approach as well as *cuSTINGER*, with a performance difference between 3× - 6×, this difference is 1.3× - 2.6× compared to *GPMA* and 0.85× - 9.7× to *Hornet*.

4.3.5 Vertex Updates

Most other dynamic graph frameworks, such as *cuSTINGER*, *aimGraph*, *GPMA* and *Hornet*, are only *partially-dynamic*. Their *SOA* approach for vertices makes it difficult to efficiently update vertices. Contrary, *faimGraph*'s *AOS* approach and index queues allow for fully dynamic vertex insertion as well as deletion. Only very recently, in 2020, *hashGraph* [3] by Awad et al. introduced another fully-dynamic framework.

4.3.5.1 Vertex Insertion

Table 4.1 and Table 4.2 show the timings for vertex insertion with a batch size of 100.000. Figure 4.14 shows update timings for a selection of graphs for batch sizes of 1000, 10 000 and 100 000. The actual insertion process for all tested graphs stays below 1 ms, the overall timing with duplicate checking stays below 50 ms for all tested graphs. Although our *reverse* duplicate checking increases performance significantly, duplicate checking still forms the bottleneck for larger graphs. As the duplicate checking involves all graph vertices, the execution time is proportional to the number of graph vertices. For the tested graphs, *faimGraph* can facilitate 2–50 million vertex insertions per second.

4.3.5.2 Vertex Deletion

Vertex deletion is more complicated than vertex insertion, as all references to the vertex in the graph must also be deleted. Table 4.1 and Table 4.2 show the performance numbers for vertex deletion in case of *undirected graphs* (UD) as well as *directed graphs* (D). In

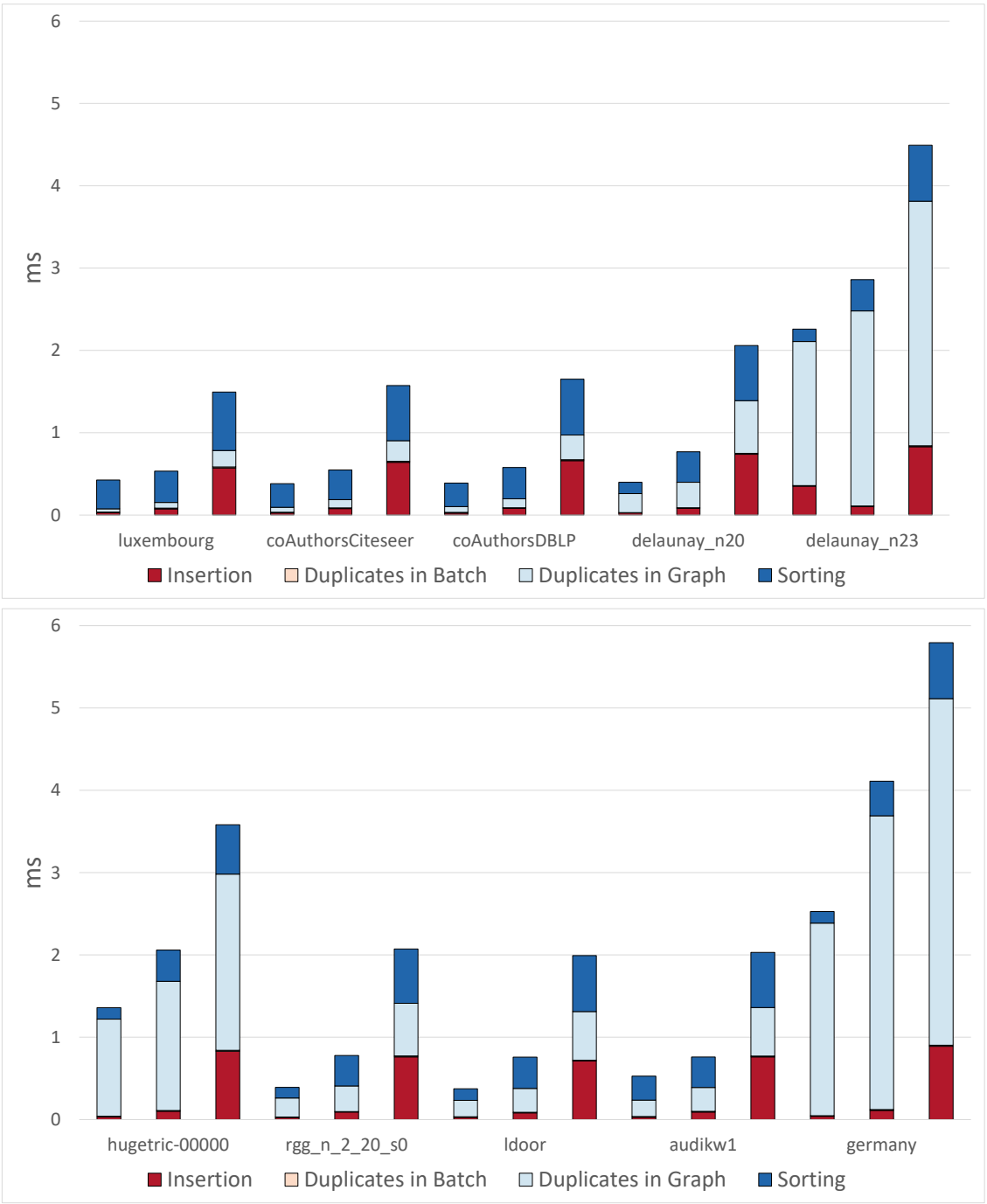


Figure 4.14: Vertex Insertion for batchsizes 1000, 10 000 and 100 000.

both cases the same graphs are used, i.e., undirected graphs can be treated as directed graphs. The performance difference is only due to the different deletion strategies that can be employed for the two cases. For undirected graphs, the vertex mentions are deleted

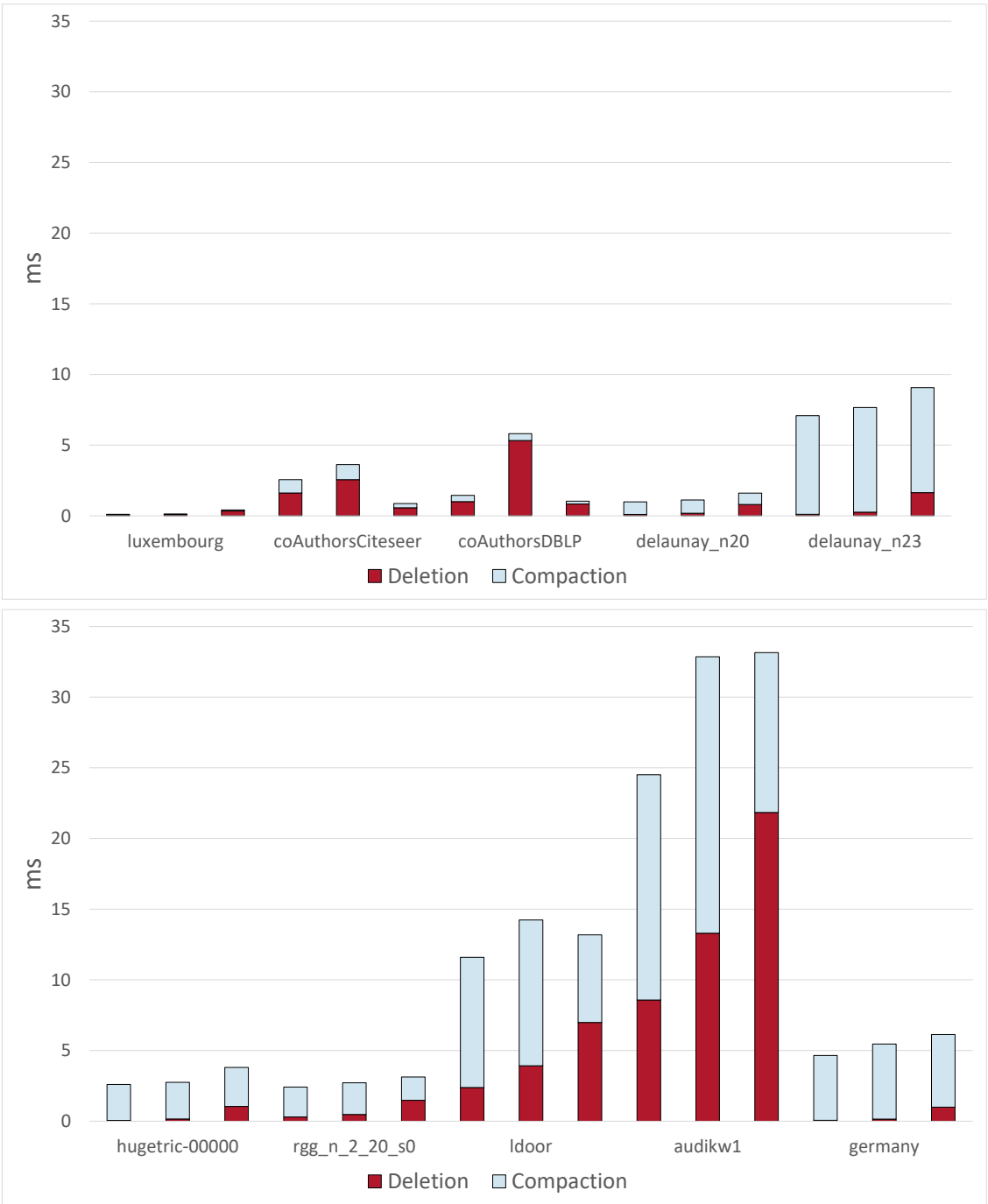


Figure 4.15: Vertex Deletion for undirected graphs for batchsizes 1000, 10 000 and 100 000.

directly in the deletion kernel, which prolongs the vertex deletion stage but does not require an additional stage afterwards. For *directed* graphs there is an extra step involved, as it is not directly obvious where the directed edges might reside in memory. This additional

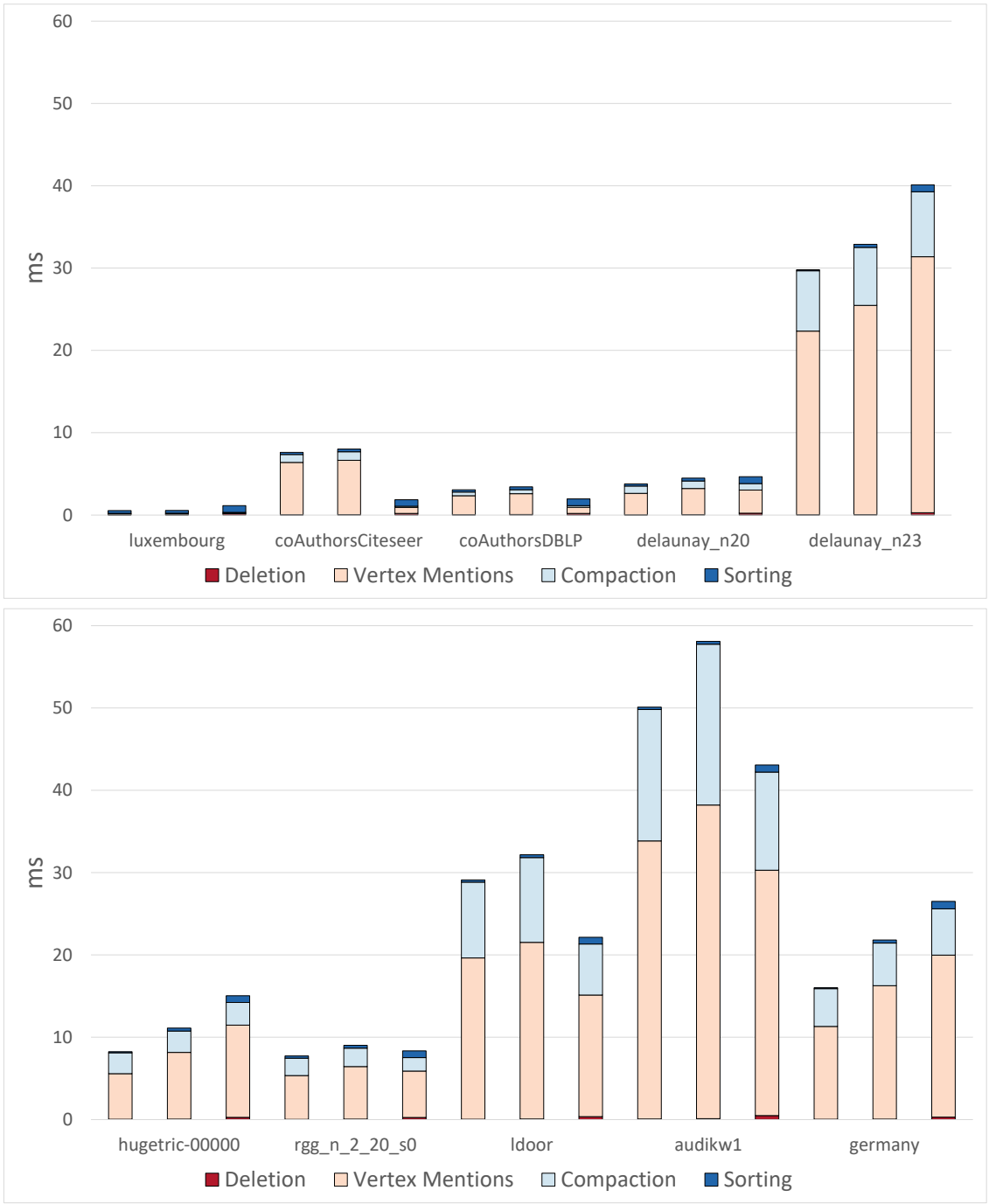


Figure 4.16: Vertex Deletion for directed graphs for batchsizes 1000, 10 000 and 100 000.

kernel once again profits from sorting the updates to utilize the reverse search pattern used to detect now invalid edges. The main difference can be observed for larger and denser graphs, as the search for references to deleted vertices is more costly as all vertices and

their adjacencies have to be checked. Overall, the framework is able to handle between 1–50 million vertex deletions per second, performance scales with both the number of vertices and edges present.

4.4 Algorithms

To evaluate the impact of our memory management data structure on algorithmic performance, we implemented triangle counting and PageRank [12] as two challenging algorithms on top of *faimGraph*. We compare our implementation to *cuSTINGER*, which includes the fast triangle counting by Green et al. [23] and a custom PageRank implementation, as well as to *Hornet*. Unfortunately, *cuSTINGER*'s implementations did not run on recent hardware, thus we additionally include performance measurements for an NVIDIA GTX 780, as can be seen in Table 4.3 and Table 4.4.

4.4.1 Work-balancing

One of the issues for graph frameworks is varying sparsity over the whole graph. Algorithms traversing these adjacencies may show significant imbalances. Thus, in addition to naïve implementations of the two algorithms, we introduce a work balancing scheme that, instead of launching a worker per vertex, calculates an offset scheme to locate individual

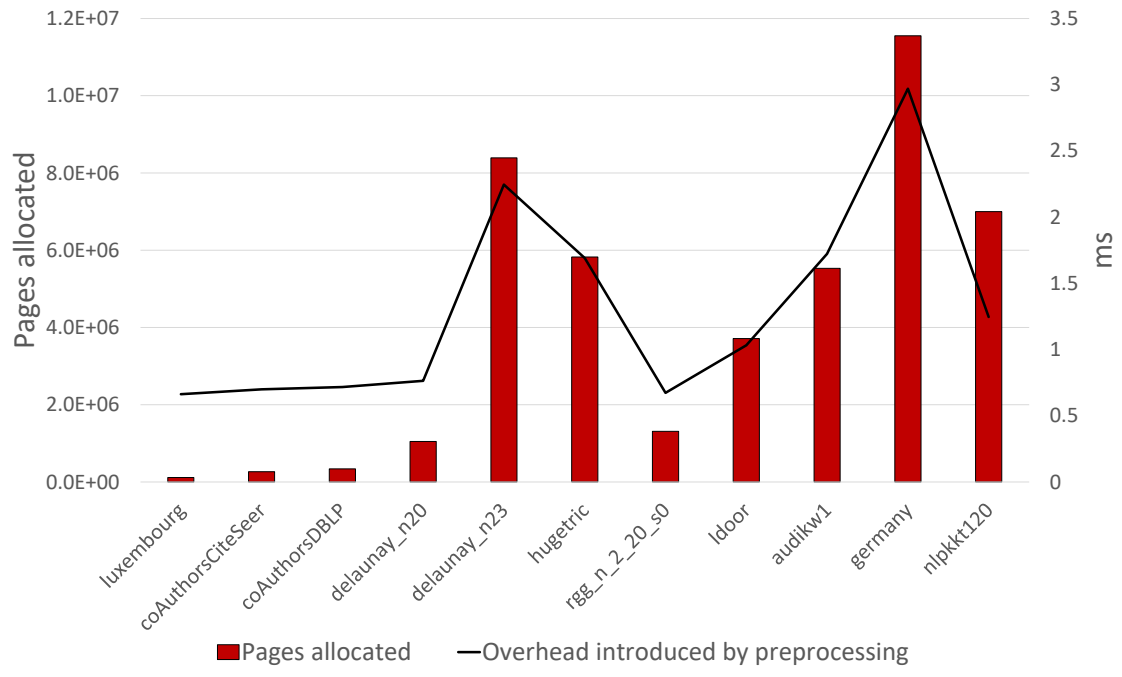


Figure 4.17: Work Balancing timing in relation to the number of pages allocated for each graph.

pages in memory. This information is used to start one worker (thread/warp/block) per page per vertex. There is a clear correlation between the overhead introduced and the pages in memory as can be seen in Figure 4.17. However, according to our experiments, the overhead stays small (between 0.5–3 ms in our tests) and the benefits drastically increase with more pages in memory.

4.4.2 Static Triangle Counting - STC

The fast triangle counting algorithm [23] employed by *cuSTINGER* is based on a list intersection algorithm called **Intersect Path**. The algorithm operates on two stages of parallelism. The first stage balances the vertices on the multiprocessors and the second stage balances the adjacency access using different block sizes. The key search strategy of the algorithm is that a sorted adjacency allows for efficient binary search to identify triangles. *cuSTINGER* includes this implementation for its own data structure and for *CSR*.

Our naïve *faimGraph* implementation starts one worker for each vertex and iterates over the respective adjacency. It then checks for each pair of vertices in the adjacency, if this pair is connected. This checking stage is only performed, if the vertex, whose adjacency is examined, has the largest index in the triple under investigation. If a triangle is found this way, the triangle count is increased for all three vertices. By assuming a

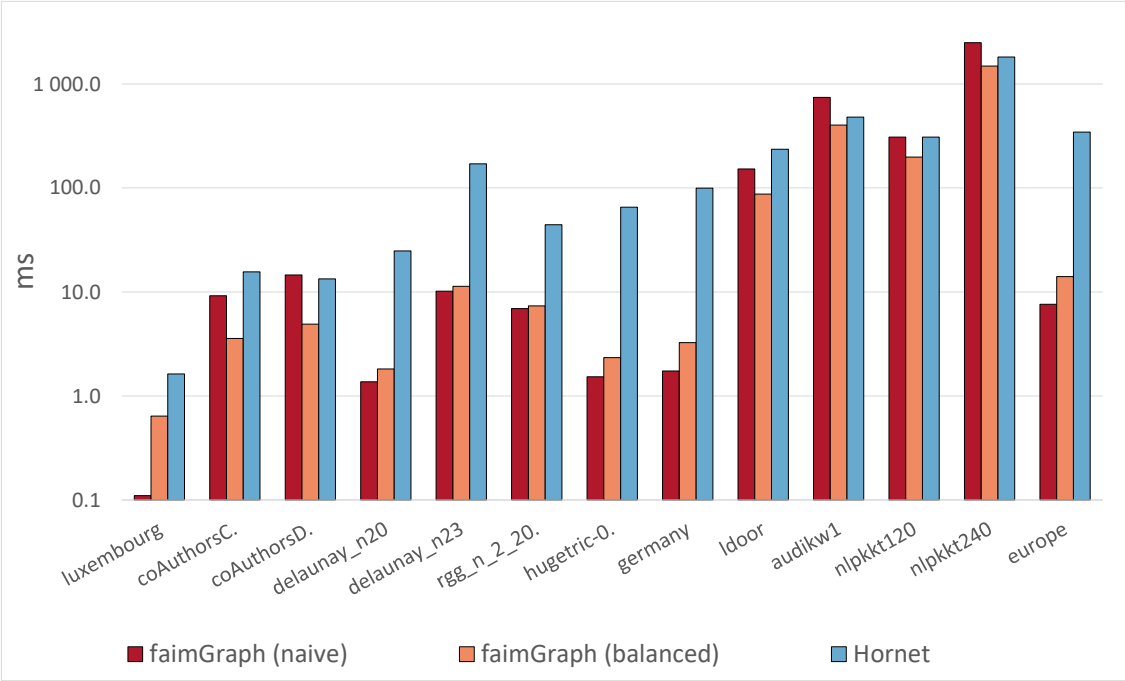
	hugetr.	*germany*	*luxemb.*	*europe*	*del*_n23*	*del*_20*	*coAuth.*
Type	Simu.	Road	Road	Road	Triang.	Triang.	Citat.
—V—	5.82M	12M	115k	50.91M	8.38M	1.04M	227k
—E—	8.73M	24.74M	239k	108.1M	25.16M	3.14M	815k
avg. deg(V)	1.50	2.06	2.08	2.12	3.00	3.02	3.59
STC							
faimG. naïve (780)	**5.21**	**5.16**	**0.077**	—	**35.56**	**5.55**	21.36
faimG. bal. (780)	7.24	8.37	0.70	—	44.85	6.14	**9.08**
cuSTINGER (780)	535.27	947.45	9.71	—	983.14	124.41	35.83
CSR (780)	7.42	698.38	7.42	—	771.06	98.59	20.94
faimG. naïve (Xp)	**1.53**	**1.74**	**0.03**	**7.60**	10.18	**1.37**	9.19
faimG. bal. (Xp)	2.34	3.26	0.64	14.04	11.34	1.82	**3.57**
Hornet (Xp)	65.26	99.52	1.63	344.30	170.66	24.78	15.61
PageRank							
faimG. naïve (780)	8.39	**8.31**	**0.54**	—	**14.28**	**1.97**	1.59
faimG. bal. (780)	**7.99**	16.16	1.09	—	16.42	2.38	**1.22**
cuSTINGER (780)	26.21	47.96	0.88	—	42.43	6.06	1.94
faimG. naïve (Xp)	3.95	5.31	**0.28**	18.49	7.07	0.99	1.09
faimG. bal. (Xp)	**3.10**	**4.74**	0.30	**17.63**	**4.79**	**0.65**	0.32
Hornet (Xp)	3.76	5.86	0.48	23.60	5.63	0.80	**0.28**

Table 4.3: Algorithmic performance in ms for *cuSTINGER*, *Hornet* and *faimGraph* (graph ordering identical to Table 4.1) with a NVIDIA GTX 780 and a NVIDIA TITAN X(p).

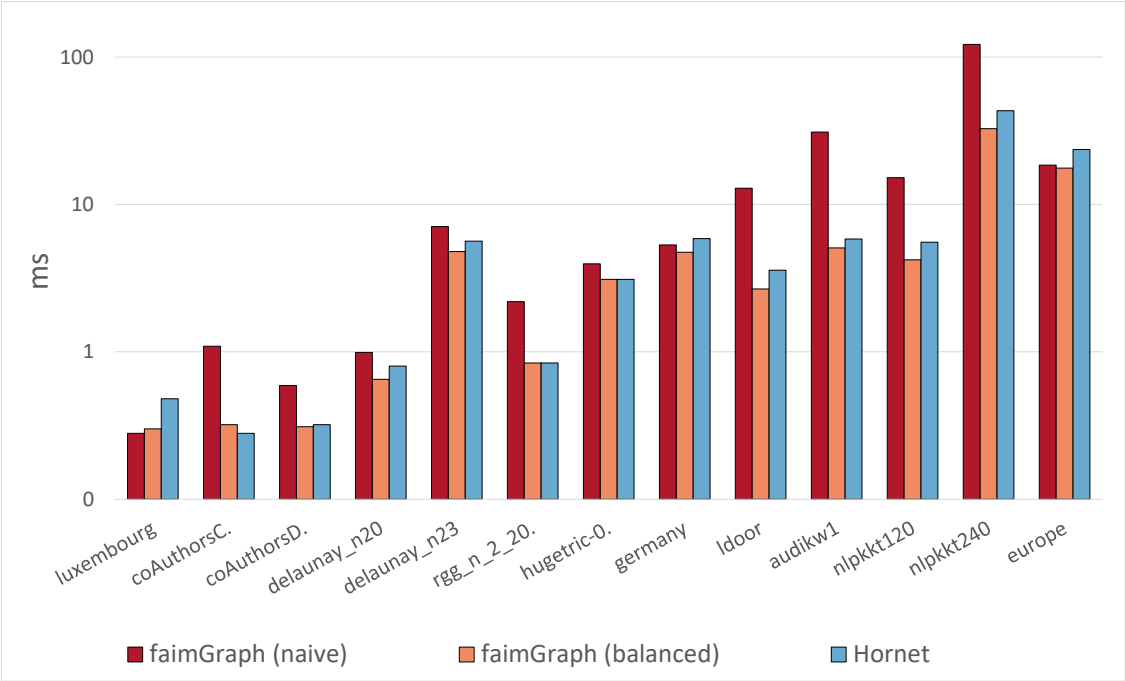
	coAuth.	*rgg_n.*	*nlpkkt200*	*nlpkkt120*	*nlpkkt240*	*ldoor*	*audikw1*
Type	Citat.	Geom.	Matrix	Matrix	Matrix	Matrix	Matrix
—V—	299k	1.04M	16.24M	3.5M	27.99M	952k	943k
—E—	1.95M	6.89M	431.9M	93.3M	746.4M	45.57M	76.71M
avg. deg(V)	6.52	6.63	26.59	26.66	26.67	47.87	81.35
STC							
faimG. naïve (780)	35.56	15.67	—	942.35	—	**258.4**	1055.93
faimG. bal. (780)	**11.79**	**8.18**	—	881.48	—	280.02	1001.6
cuSTINGER (780)	45.72	166.21	—	649.07	—	379	1277.1
CSR (780)	26.86	140.75	—	**327.43**	—	260.03	**635.8**
faimG. naïve (Xp)	14.56	**6.92**	1413.9	307.89	2490.4	152.41	742.23
faimG. bal. (Xp)	**4.90**	7.34	**923.98**	**197.80**	**1484.71**	**87.26**	**402.56**
Hornet (Xp)	13.36	44.26	1111.5	308.63	1812.3	235.33	479.63
PageRank							
faimG. naïve (780)	**1.01**	4.04	—	26.51	—	16.18	31.4
faimG. bal. (780)	1.36	**4.03**	—	**16.02**	—	**11.98**	**18.90**
cuSTINGER (780)	2.62	10.31	—	57.53	—	20.25	30.18
faimG. naïve (Xp)	0.59	2.19	69.32	15.20	121.91	12.87	31.00
faimG. bal. (Xp)	**0.31**	**0.84**	**18.94**	**4.21**	**32.67**	**2.67**	**5.07**
Hornet (Xp)	0.32	5.26	25.08	5.54	43.29	3.58	5.83

Table 4.4: Algorithmic performance in ms for *cuSTINGER*, *Hornet* and *faimGraph* (graph ordering identical to Table 4.2) with a NVIDIA GTX 780 and a NVIDIA TITAN X(p).

sorted (in ascending order) adjacency, this approach can even halt the procedure earlier, as soon as both vertices in the vertex pair under investigation are larger than the source vertex (in these cases a possible triangle will be entered by one of the other vertices). The balanced *faimGraph* implementation works similarly, but starts on worker per page per vertex, reducing the workload per worker. Performance numbers are recorded in Table 4.3 and Table 4.4. Both *cuSTINGER* and *faimGraph* utilize the property that the adjacency is sorted to make comparison possible. *faimGraph* is able to significantly outperform *cuSTINGER* in all but *nlpkkt120* (11), which shows very long adjacencies. *faimGraph* can only partially derive an advantage from a sorted adjacency as an efficient search within a sorted array is only possible within page boundaries. Interestingly, our work balancing also outperforms the highly compact CSR format (using the fast triangle counting algorithm) in all but three cases. *faimGraph* is not well suited for naïve random adjacency access and thus triangle counting is one of the most challenging use cases for our data structure. Hence, a straight forward translation of the *Intersect Path* algorithm to *faimGraph* would also not show the same performance results as on a simple array structure. *faimGraph* is well suited even for memory intensive algorithms, if the average adjacency size does not grow incessantly, using *work balancing* even unbalanced graphs can be handled well. Overall, in 10 out of 11 cases, *faimGraph* has a performance lead between 1.25× - 100× over *cuSTINGER*, only falling behind in one test case. Compared to *Hornet*, *faimGraph* has a performance lead between 1.19× - 57× in all cases, as can be seen in Figure 4.18a.



(a) Performance for static triangle counting (STC).



(b) Performance for PageRank

Figure 4.18: Algorithm performance comparing a naïve and balanced version of *faimGraph* against *Hornet*.

4.4.3 PageRank

PageRank [12] is a fairly straightforward algorithm. The algorithm has to traverse the adjacencies of all vertices and compute the contributions of all relationships for each vertex in push mode, similar to an *SpMV*. This means that every edge is touched exactly once, the same is true for every vertex. The only point of contention remains the *PageRank vector* itself. Table 4.3 and Table 4.4 show the direct comparison between *cuSTINGER*, *Hornet* and the two (standard and balanced) *faimGraph* implementations. As PageRank has moderate memory access requirements and does not benefit from sorting, *faimGraph* is able to outperform *cuSTINGER* in all cases due to the more efficient memory access and footprint characteristics. Unbalanced and larger graphs once again profit from *work balancing*. Overall, *faimGraph* is able to outperform *cuSTINGER* by a factor of 1.5× - 5.5×. The same is true compared to *Hornet*, but the performance difference is smaller overall in a range between 0.88× - 6.2×, as can be seen in Figure 4.18b.

4.5 Discussion

faimGraph is a memory-efficient, fully dynamic graph solution with autonomous memory management directly on the GPU. Based on a *queueing* scheme, memory is fully reused within the system, reducing memory requirements by multiple orders of magnitude in the long run as well as memory fragmentation, permitting edge insertions and deletion

	aimGraph	faimGraph	cuSTINGER	Hornet	GPMA
Memory Layout	Linked edge blocks	Linked pages	Contiguous memory	Contiguous memory	PMA-style memory layout
Adjacency Access	Fast on edge block, traversal needed	Fast on page, traversal needed	Always fast	Always fast	Need to handle “holes” due to storage
Memory Efficiency	Does not reuse memory	Reuse pages efficiently	Does not reuse memory	Reuse memory efficiently	Efficient for small graphs, leaves “holes”
Initialization	Very fast, precompute and setup in parallel	Very fast, precompute and setup in parallel	Individual allocations from CPU	More efficient than **cuSTINGER** but still CPU bound	Build complex PMA
Edge Updates	Fast for uniform updates on sparse graphs	Fast for all updates on sparse graphs, slows down for denser graphs	Fast for small allocation state changes	Fast for small allocation state changes	Fast as long as little balancing is required
Vertex Updates	**Not** supported	Both insertion and deletion supported	**Not** supported	**Not** supported	**Not** supported
Algorithms	Fast for sparse, uniform graphs, traversal a factor	Fast for sparse and unbalanced graphs, traversal a factor	Contiguous memory, but no balancing	Contiguous memory, but no balancing	Implicitly sorted, but traversal is challenging

Figure 4.19: Overview comparison of *aimGraph*, *faimGraph*, *cuSTINGER*, *Hornet* as well as *GPMA*.

according to arbitrary patterns. Thus, *faimGraph* can be safely used in real-world scenarios without threatening system failures due to out of memory. Furthermore, *faimGraph* is fully dynamic, allowing for efficient vertex insertion and deletion at high rates, increasing access characteristics by efficiently reusing free vertex indices. Our *vertex-centric* update scheme allows lock-free edge updates, which increases performance by one order of magnitude under high update pressure. Edge updates can also respect sort order with little overhead.

faimGraph outperforms the previous state-of-the-art in all tested graphs in terms of edge update rate (up to 150× higher update rate) as well as initialization time (up to 300× faster). The framework can hold tens of millions of vertices and hundreds of millions of edges in memory. It is able to process up to 200 million edge updates and more than 300 million adjacency queries per second for the tested graphs. Vertex updates can also reach between 1–50 million updates per second. To validate algorithmic performance of our data structure, we tested triangle counting and PageRank. Although *faimGraph* uses a more complicated data structure to allow for memory reclamation, it performs surprisingly well for the random access heavy triangle counting, outperforming *cuSTINGER* in all but one case. For PageRank, *faimGraph* showed the best performance in all cases.

For the first time, it is possible to perform both in parallel using autonomous memory management; we even support algorithms to directly manipulate the graph. However, both operations may still require communication and appropriate synchronization, which poses new scheduling challenges which we leave as future work. Nevertheless, we believe that *faimGraph* is a first big step towards using GPUs for real-world, dynamic graph processing.

5

Ouroboros - Virtualized Queues for Dynamic Memory Management on GPUs

Contents

5.1 Introduction

Dynamic graph management on the GPU, as possible with *aimGraph* and *faimGraph*, is essentially one application of general purpose memory management on the GPU. *faimGraph*, at its core, could already be used to efficiently allocate pages of a fixed size directly on the GPU, even lacking a graph context. This is possible thanks to the built-in memory re-use queues, which can hold previously allocated edge blocks or pages to be used by later allocations. All of this is possible directly on the GPU without any CPU synchronization.

While the core idea is already sound and translates well to general purpose memory management, typically one requires more than just a single available page size to support applications with reasonable memory fragmentation. As is common and other designs already show, such a memory manager should be able to return arbitrary page sizes to the user. These can be limited in size and aligned to some intermediary steps, but a single size certainly does not suffice. To solve this issue, an initial step involving the design of

faimGraph could utilize a memory manager in charge of multiple re-use queues, each of these supplying one page size supported in the system. As most other memory managers on the GPU, one could choose one large chunk size, which can then be split into smaller pages servable to the user, with one queue per page size. Leaving aside the access and chunk/page allocation intricacies, this still would potentially entail significant overhead, as static allocations required for the queues would have to be present even if not needed at all times. Especially when supporting a larger range of allocations as well as allocation sizes, these re-use structures would incur significant static overhead. This restricts the possible memory allocation size of an application unnecessarily.

In this (r)evolution of the base design, we address these concerns and present a general purpose dynamic memory manager for the GPU. Using the memory manager and queues established with *faimGraph* and introducing new ideas to virtualize these queues for memory efficiency, a new concept for dynamic resource management in general as well as dynamic graph management in particular is born. We provide overall six different variants of this memory manager with varying degrees of memory efficiency and access performance that can be customized for specific use cases. In the following sections, we will discuss the intricacies of this design, called *Ouroboros*. We will present a more general approach in describing the pure memory management capabilities, which are usable in any scenario requiring dynamic resources on the GPU. This includes two base variants managing either pages directly or chunks with free pages on them. Building on these designs, we present two virtualization techniques to rigorously reduce the memory overhead while keeping most of the performance benefits. We also evaluate its allocation capabilities as well as memory efficiency and real-world performance compared to all other publicly available memory managers on the GPU, including *XMalloc*, *ScatterAlloc*, *Halloc*, *RegEff* and the *CUDA-Allocator*. As its basic structure still lends itself to dynamic graph management as well, this discussion is deferred to Chapter 6.

5.2 Building Blocks and Background

In the following, we discuss the building blocks essential to *Ouroboros* and their relevance to the system.

5.2.1 Memory Management

Dynamic memory management on the GPU presents a number of challenges:

1. The high number of concurrently active threads on a modern GPU can result in an equally high number of concurrent allocation/deallocation requests. On modern GPUs, this can be as high as 172 k simultaneously active threads (on the NVIDIA GV100 architecture).

2. Data structures and access primitives have to be able to deal with such pressure.

3. Since memory is a scarce resource in the context of GPUs, keeping memory fragmentation to a minimum and memory must be used efficiently.

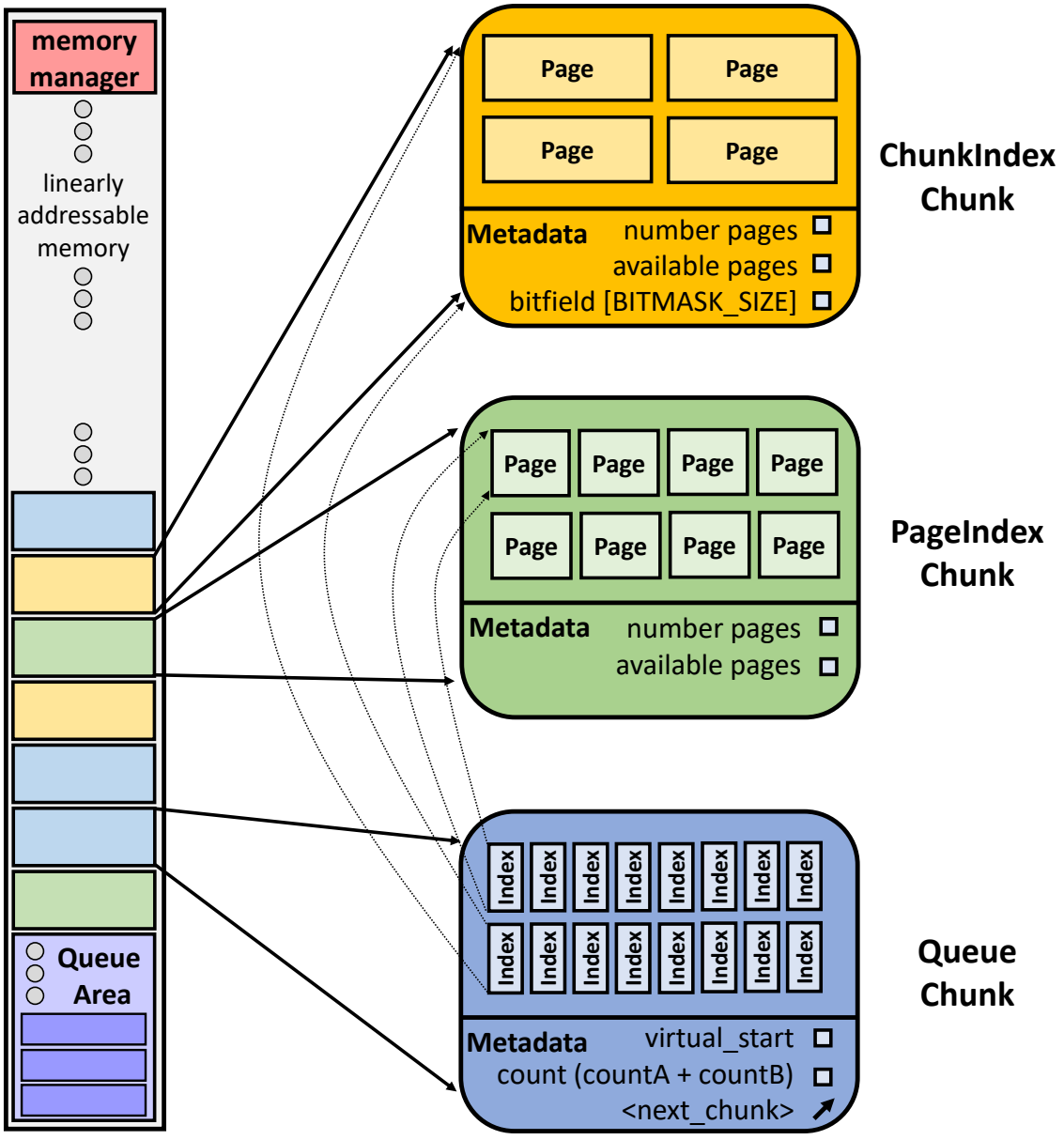


Figure 5.1: One large memory allocation is split into equally-sized chunks, which are interpreted as *ChunkIndex-Chunk* (referenced as whole chunk for reuse), *PageIndex-Chunk* (referenced per page) or as *Queue-Chunk* (holds chunk/page indices). The beginning holds the memory manager itself and the memory right after can be linearly addressed by an application. The end holds the base structure for potentially many queues for chunk/page reuse.

4. The system should not force a processing model on the user, but allow single threads to allocate new memory.

Similar to other allocators (including *CUDA-Allocator*), *Ouroboros* starts by allocating a block of memory to be managed on the GPU. The size of this block can be set heuristically to fit a given problem domain, can be provided by the user or even encompass the whole GPU memory. Should the given allocation be too small, the system has the option to automatically re-initialize in a larger area of memory. This is possible as all information about memory resources within *Ouroboros* is relative, hence no updating of many references would required in the case of re-initialization. All allocation requests are handled directly on the GPU, avoiding costly round-trips to the CPU, keeping this GPU-autonomy established with *aimGraph* and *faimGraph*. A *memory manager* keeps track of all available resources and offers standard *malloc* and *free* functionality for individual threads. These calls either return/receive pointers directly as is usual but can also operate directly on smaller indices for more efficient storage overall.

The designated dynamic memory is subdivided into equally-sized *chunks*, which can be allocated from the *memory manager* in *O(1)* using an `atomicAdd()` on a global offset. The size of these *chunks* can be matched to the specific application (standard size is 8 KiB which suffices to handle all graphs within the *DIMACS10* [5] graph dataset). This determines the maximum allocation size of the base instance of *Ouroboros*. To service larger allocations, multiple instances of *Ouroboros* with multiples of the base *chunk* size can be combined. Alternatively, a second allocator, like the library-provided *CUDA-Allocator*, which offers standard *malloc* and *free* operations on the GPU, can be used or *Ouroboros*'s chunks could be integrated into the page-table system (by a vendor).

5.2.2 Chunks

The dynamic memory region used by the *memory manager* is split into equally-sized *chunks* of memory (larger instances use multiples of this chunk size). Chunks are addressed by an integer index, enabling efficient re-initialization in a different memory space. Each chunk consists of a small region for *meta data* (padded to multiples of the cacheline size (128 B)) and a large region to hold data (both specific to the actual use case, typically the *meta data* easily fits into (128 B, leaving (8064 B for user data for 8 kB chunks). Since each chunk has the same size, they can be addressed by a plain index. This relative indexing also enables efficient re-initialization in a different memory space, if needed, as only the global offset has to be updated in a larger (or smaller) region. Chunks are used in three different ways, as shown in Figure 5.1:

- **ChunkIndex-Chunk** stores user data on pages. If pages become free on this chunk, the chunk index is placed in a queue, a *bit-field* is used to manage the allocations on the chunk. The *bit-field* size depends on the maximum number of pages in which a chunk can be split. A chunk can be allocated to a different page size or different chunk type if completely freed.

Algorithm 5: Basic enqueue and dequeue functionality

```
1  Function enqueue(index)
2      if atomicAdd(fill_count, 1) > size then
3          atomicSub(fill_count, 1)
4          return
5      pos ← atomicAdd(back, 1) mod size
6      while atomicCAS(q[pos], del, index) ≠ del do
7          sleep()
8  Function dequeue(index&)
9      if atomicSub(fill_count, 1) ≤ 0 then
10         atomicAdd(fill_count, 1)
11         return
12     pos ← atomicAdd(front, 1) mod size
13     while index ← atomicExch(q[pos], del) = del do
14         sleep()
```

- **PageIndex-Chunk** stores user data on pages. Page indices are directly stored in *queues* for reuse. It retains a specific page size once set.

- **Queue-Chunk** is used as index storage for *virtualized queues*, storing queue data. It can also be allocated to a different use case once empty. It holds $ChunkSize/sizeof(IndexType)$ indices.

Chunks used for user data are split into equally-sized *pages*. The largest page size is limited by the chunk size, whereas each split halves the page size. The number of differently-sized pages available in the system determines the number of *queues* required for potential reuse of pages, e.g.,, a chunk size of 8 KiB and ten queues allow for allocations in the range of 16 B–8192 B within one instance of *Ouroboros*. To reduce fragmentation, previously allocated, now empty chunks are held in an index queue, which allows the memory manager to efficiently reuse empty chunks before allocating new chunks. This way overall fragmentation is reduced.

5.2.3 Queues

We utilize an *array-based* method for memory reclamation in form of an *index queue*, similar to the queue introduced in Section 4.2.2 used within *faimGraph*. Queues are not only used for memory reuse, but also to reuse other dynamic objects, which includes *QueueChunks* and can be extended to anything else, e.g., dynamic vertex indices in a dynamic graph context. Algorithm 5 demonstrates the basic *enqueue* and *dequeue* functionality of the queues.

Both first test the *fill_count* to check the viability of the operation (*dequeue* failing is a common occurrence). Then, both use *atomic functions* to either write or read a queue value. The queues are initialized at the start-up of the system to contain *deletion markers*. This ensures that concurrent enqueues and dequeues are possible. The checks in line 6 and 13 are required, since *enqueue* might want to write to a spot, which was already advertised as free by a *dequeue* operation, but the value has not been read yet. This is being safeguarded against using an `atomic-Compare-And-Swap` operation with the deletion marker. On the other hand, a *dequeue* operation might want to read a value that has been advertised as present by an *enqueue* operation, but the write is not yet visible in global memory. The *sleep* functionality called in lines 7 and 14 are implemented using *sleep()* for GPUs with $CC \geq 7.0$ and a *threadfence()* for those lower than that. In both cases the goal is to trigger a rescheduling to potentially allow other threads to finish their operations or in general free up compute resources for other threads to use.

While this basic queue design works sufficiently well to manage simple, atomic objects (like indices of vertices or indices of pages in case of a singular page size), this design needs further attention to also work in the context of batch-based allocation as is the case with us splitting larger chunks into pages. A necessary part of this approach is a new synchronization primitive, which is discussed in the next section.

5.2.4 Access primitive

In order to regulate access to enqueues/dequeues, we use an *access primitive* that keeps track of the total number of pages in the queue and can be used to drop the *fill_count* in most cases. As access primitive we use a *bulk semaphore* [20]. It enables a scalable, two-stage resource management system, which is based on three counters:

- **count (C)**: Pages currently available
- **expected (E)**: Pages expected to become available
- **reserved (R)**: Pages reserved by waiting threads

It improves upon a simpler *counting semaphore*, which automates the process of delegating which threads actually allocate a larger resource to deal out shares to other waiting threads, by interleaving the allocations more efficiently. This allows potential allocating threads to start much earlier, leading to improved batch allocation performance. The *expected availability* is defined as the value after all *expected* pages have been added to the *existing* pages and all *reserved* pages have been subtracted. Based on this value, each thread determines if it can fulfill its allocation request, if it has to allocate a new *chunk* of pages first or if it can reserve a page on a chunk that is currently being allocated. The bulk semaphore implements two functions, outlined in Algorithm 6:

- $wait(N, \#pages, allocFunc())$: try to allocate N pages. If *expected availability* is $< N$, allocate a new chunk with $\#pages$ pages using *allocFunc()*. If the current

count is large enough, decrement and continue. Otherwise increase the *reserved* value and wait for the resources to be allocated.

- *signal*$(N, \#pages)$: free up N pages by increasing *count*, if $\#pages > 0$ reduce *expected* by $\#pages$.

Our implementation packs all three counters into one 64 bit value, resulting in 21 bits per counter. Simple manipulation of counters (as in lines 2, 19 and 22) can be done with one single atomic operation. Only lines 8 to 16 in Algorithm 6 are implemented using an `atomicCAS` operation, since multiple comparisons and assignments have to be performed atomically. To this end, the value is read from memory, its internal counters are checked and modified accordingly and the atomic operation is used to compare the value in memory to the previously read value. Only if they match (no change has happened), the new value is written to global memory, otherwise the operations are repeated with the

Algorithm 6: Access primitive functions

```
1  Function Sem::wait(N, #pages, allocChunk())
2      atomic
3          if Sem.C ≥ N then
4              Sem.C ← Sem.C − N
5              return
6          Sem.C ← Sem.C + N
7      while True do
8          atomic
9              if Sem.C + Sem.E − Sem.R < N then
10                 Sem.E ← Sem.E + #pages
11                 allocChunk()
12             else if Sem.C ≥ N then
13                 Sem.C ← Sem.C − N
14                 return
15             else
16                 Sem.R ← Sem.R + N
17         while Sem.C < N and Sem.R < (Sem.C + Sem.E) do
18             sleep
19         atomic
20             Sem.R ← Sem.R − N
21 Function Sem::signal(N, #pages)
22     atomic
23         Sem.C ← Sem.C + N
24         Sem.E ← Sem.E − #pages
```

new value. As far as we know, this is also the first publicly-available implementation of the *bulk semaphore*, which also includes a variant, which also works (though less efficient) on *GPUs* prior to the *Volta* generation with *Independent Thread Scheduling (ITS)*. For a more detailed explanation as well as an evaluation comparing this to counting semaphores consult Gelado et al. [20].

5.3 Queues for Memory Management

The most basic tools for memory management are efficient *index queues* [26, 62]. These are usually implemented as *array-based* queues, operating on top of a ring buffer, as already discussed in Section 5.2.3. Concurrent access and efficient queries for empty states are realized using a *front* and *back* pointer as well as a *fill count*. While these queues can efficiently manage indivisible objects, they are unsuitable for handling chunks split into pages, i.e., batch-based allocation. Furthermore, they do not provide an efficient mechanism to allocate new pages/chunks once the queue is empty. We propose two different evolutions of this queue, utilizing the *bulk semaphore* for efficient allocation. This design reduces fragmentation and always allocates the least amount of memory possible, as previously deallocated memory is used before new memory is allocated by the memory manager. An instance of *Ouroboros* can be configured with queues managing either pages (discussed in Section 5.3.1) or chunks (containing available pages, discussed in Section 5.3.2), as can be seen in Figure 5.2. One instance manages one or the other, but *Ouroboros* can combine multiple instances of itself with different managing capabilities. Hence, one could choose one instance of *Ouroboros* managing pages for smaller allocations (as page-based management is faster) and one instance managing chunks for large allocations (which are more memory efficient but slower, but as large allocations are less likely this is less of an issue).

5.3.1 Queues managing pages

This queue type is the most straightforward evolution of the *index queue*, as it also stores page indices directly. The main difference can be found in the access management. The fill counter is replaced by a *bulk semaphore* to allow for efficient batch allocation of pages. This queue offers *O(1)* allocation (dequeue from the queue), as long as the queue still holds free pages, and *O(1)* deallocation (enqueue into the queue), as long as the queue is not full. Once the queue is empty, the *bulk semaphore* allows for efficient and interleaved allocation of new pages from chunks. Algorithm 7 lists the high-level steps needed for page allocation and deallocation. In the allocation stage, a thread first interacts with the *bulk semaphore*, calling *wait*() (line 11) with the option to allocate a new chunk of pages (*allocChunk*() in line 1). If a thread is designated to allocate new pages, the allocation is first signalled to the *bulk semaphore* before the pages are added to the queue. If a page is available (indicated by the *bulk semaphore*), the corresponding position is determined and

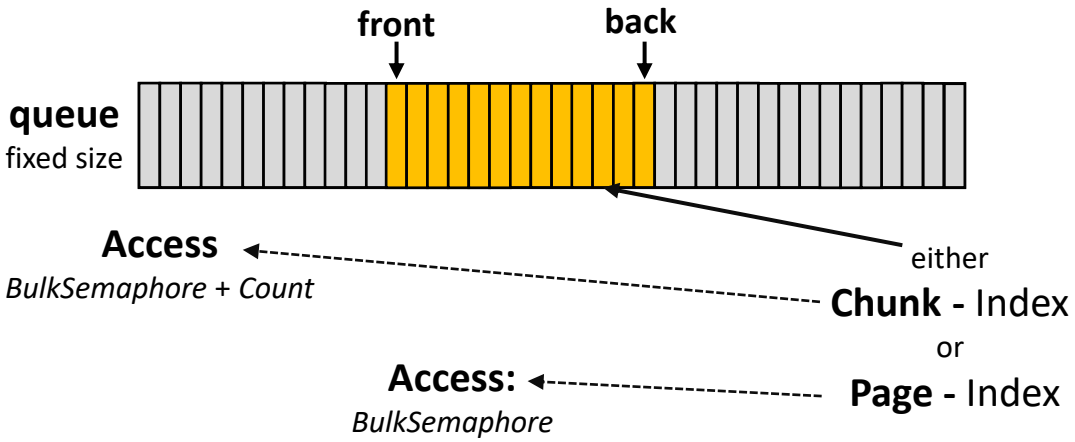


Figure 5.2: A *standard* queue can either manage pages directly or chunks with free pages on them. The access management is either just a *bulk semaphore* in case of *page-based* queues and are augmented by a *fill count* in case of managing chunks.

the page index read, as detailed in Section 5.2.3. Deallocation works exactly as *dequeue*() described in Algorithm 5, replacing the $fill\_count$ with a *bulk semaphore*.

This design excels in terms of allocation speed, but bears some minor disadvantages.

Algorithm 7: Allocate / Free page with the page-based queue

```
1  Function allocChunk(memory_manager, #pages)
2      if sem.signal(#pages, #pages) < #spots then
3          memory_manager.allocChunk(index)
4          pos ← atomicAdd(back, #pages)
5          foreach page in chunk do
6              index ← createIndex(chunk, page)
7              while atomicCAS(q[pos], del, index) ≠ del do
8                  sleep()
9              pos ← (pos + 1) mod size
10 Function allocPage(memory_manager, index&)
11     sem.wait(1, #pages, allocChunk)
12     pos ← atomicAdd(front, 1) mod size
13     while (index ← atomicExch(q[pos], del)) = del do
14         sleep()
15     return memory_manager.getPage(index)
16 Function freePage(index)
17     if sem.signal(1, 0) < #spots then
18         q.enqueue(index)
```

One drawback is limited chunk re-usability. Once assigned to a page size, a chunk cannot be assigned to a different page size or chunk type. Even if all pages of a chunk are free and thus currently in the queue, chunk reuse would require removing all its page indices from the queue. Additionally, each free page occupies an entry in the queue. Thus, potentially larger queue sizes are required to handle large allocation fluctuations. Consequently, the allocation from the chunk pool takes more time, as all pages are added to the queue by one thread.

In most cases, neither is a critical issue. Allocation sizes typically fall into a certain range and do not sweep from small to large, putting the chunk re-usability into perspective. Furthermore, especially considering our virtualization efforts discussed later, memory overhead also is not the main concern for many scenarios. Overall, the *page-based queue* is the clear performance choice with great user-memory efficiency and acceptable memory overhead. *Ouroboros* using this design is henceforth referred to as *Ouro-S-P*.

5.3.2 Queues managing chunks

One way to overcome the aforementioned issues is to store *indices of chunks* with free pages directly in the queue. No matter if one or all pages are free on a chunk, it always occupies just a single queue spot. On average, this reduces the required queue size substantially. In the worst case, if only a single free page is left on each chunk, it needs as much space as the page index queue. Furthermore, allocation of a new chunk results in a single enqueue into the queue instead of one per page.

Contrary to the aforementioned queue design, both the fill counter and the *bulk semaphore* are needed here. The fill counter reflects the number of chunks in the queue while the *bulk semaphore* keeps track of the total number of free pages on those chunks. The allocation procedures, shown in Algorithm 8, differ from the previous, simpler approach. Allocation follows a two-stage approach. First, the semaphore is queried as before, but this call only guarantees a successful allocation but does not give more information about where to locate the actual page. To locate the actual page, the current front pointer is loaded and the chunk at this position is queried for a free page, see Algorithm 9. In case of failure, the current thread advances in the queue to the next chunk and repeats the query until successful. This query is a simply *atomic* test on the `count` variable of a *ChunkIndex-Chunk*. Once all pages on a chunk have been allocated, it can be removed from the queue (line 18 in Algorithm 8). As multiple chunks may become empty simultaneously, the $front$ index is advanced to the largest of these; the chunk is removed from the queue and the fill count is reduced. A re-enqueue (line 15 in Algorithm 8) is needed as a chunk may be emptied (which means one thread will dequeue it from the queue), but shortly after, threads free pages on it (which should result in this chunk being added again) without noticing the prior removal, see line 2–14 in Algorithm 9. In this case it might fall to one allocating thread to re-enqueue the chunk.

Deallocation usually only signals the arrival of a new page and sets the corresponding

bit in the chunk's bit-mask. The first deallocation on a chunk adds the chunk to the queue (line 28 in Algorithm 8). The last deallocation on a chunk (line 30 in Algorithm 8) tries to reduce the semaphore value by a full chunk capacity. If successful, it flashes the bit mask of the chunk using *atomicCAS* operations. If this succeeds as well, the chunk is removed from the queue and can be reused as any chunk type. Due to the deletion

Algorithm 8: Allocate / Free page with the chunk-based queue

```
1  Function allocChunk(memory_manager, #pages)
2      memory_manager.allocChunk(index)
3      q.enqueue(index)
4      sem.signal(#pages, #pages)
5  Function allocPage(memory_manager, index&)
6      sem.wait(1, #pages, allocChunk)
7      pos ← front mod size
8      while True do
9          chunk_index ← q[pos]
10         if chunk_index ≠ del then
11             chunk ← getChunk(chunk_index)
12             mode ← chunk.allocPage(index)
13             if mode = SUCCESS then
14                 break
15             else if mode = RE_ENQUEUE then
16                 q.enqueue(chunk_index)
17                 break
18             else if mode = DEQUEUE then
19                 atomicMax(front, pos + 1)
20                 atomicExch(q[pos], del)
21                 atomicSub(count, 1)
22                 break
23         pos ← pos + 1 mod size
24     return memory_manager.getPage(index)
25 Function freePage(index)
26     chunk ← getChunk(index.chunk)
27     mode ← chunk.freePage(index)
28     if mode = FIRST_FREE then
29         q.enqueue(index.chunk)
30     else if mode = DEQUEUE then
31         atomicExch(q[chunk.queue_pos mod size], del)
32         chunkQueue.enqueue(index.chunk)
33     sem.signal(1, 0)
```

Algorithm 9: Allocate page from chunk

```
1  Function Chunk::allocPage(index)
2      while curr_count ← atomicSub(count, 1) ≤ 0 do
3          if curr_count ← atomicAdd(count, 1) < 0 then
4              return CONTINUE_TRAVERSAL
5          else if curr_count = 0 then
6              mode ← RE_ENQUEUE
7      if curr_count = 1 then
8          if mode = RE_ENQUEUE then
9              mode ← SUCCESS
10         else
11             mode ← DEQUEUE
12     else
13         if mode ≠ RE_ENQUEUE then
14             mode ← SUCCESS
15     mIndex ← 0
16     while True do
17         mask = bitmask[mIndex]
18         while lowestbitset ← findFirstSet(mask) do
19             bits ← createPattern(lowestbitset)
20             mask ← atomicAnd(mask[mIndex, bits])
21             if checkBitSet(mask, lowestbitset) then
22                 return mode
23         mIndex ← (mIndex + 1) mod maskSize
```

marker, the queue location is simply skipped during allocation. A threshold can be set to keep a certain percentage of chunks in the queue, as allocations from an empty queue are more expensive as from a queue already holding indices.

The clear focus of this queue type is memory efficiency, as it requires less queue storage on average. Furthermore, as chunks can easily be removed from the queue, they can also be used for different purposes within the system again. Comparing performance to the *page-based* variant, the two-stage approach will typically perform worse. This is due to the two-stage access design. An index into the queue for the *page-based* queue is automatically the page that is returned to the user. In the worst case scenario, a slight wait time is introduced by waiting on this spot to become available. With the *chunk-based* design, an index into the queue only shows a potential candidate for the second stage of the allocation. In case the current front will not be advanced fast enough (line 19 in Algorithm 8) given a high number of concurrent threads, this can lead to queue traversal. Furthermore, allocation on a chunk also is more involved as a free bit has to be located

in the bit mask, which once again is not trivial in a highly concurrent environment with potentially hundreds of threads trying to allocate a page on a chunk.

Overall, *chunk-based* queues perform well enough and particularly excel at memory efficiency. Hence, they lend themselves especially for larger instances of *Ouroboros*, as described earlier, where smaller, frequent allocations are handled by *Ouro-S-P* and larger, less frequent allocations are handled by *chunk-based Ouroboros*, henceforth referred to as *Ouro-S-C*.

5.3.3 Supporting different allocation sizes

Each queue described so far is built to handle pages with the same size. For each page size, a queue must be instantiated in memory. Since memory is a precious resource, keeping this overhead low is crucial and a lot of applications (e.g., dynamic graphs) require more and more of it. Furthermore, the queue capacities may have to be large to hold the desired number of re-usable items: Consider the example of a dynamic graph where one million vertices require reallocation, freeing and allocating one million pages. All freed pages might end up in one queue and all allocated come from another single queue, meaning all queues require the capacity of one million.

5.4 Virtualized Queues for Memory Management

The aforementioned queues potentially suffer from significant memory overhead. To support a multitude of different page sizes in systems with significant reuse, the memory overhead can become prohibitive.

We introduce two variants of our queue-based memory management system, which reduce these overheads by virtualizing the base queue and thereby only keep the currently required queue size allocated in memory. Queue data itself is stored on *QueueChunks*, allocated directly from the memory manager or from the *chunk reuse queue* in *O(1)*. Once all elements on a specific *QueueChunk* are freed, i.e., dequeued, it is placed in a *chunk reuse queue* to be reused later, potentially as a different chunk type, further reducing potential fragmentation. This reduces the overall memory requirements drastically compared to the statically sized queues, greatly improving the suitability of this system to even large use-cases like dynamic graph management.

5.4.1 Virtualized Array-Hierarchy Queue (*VAQ*)

A *VAQ* replaces statically allocated queues by a much smaller chunk pointer queue (Figure 5.3). Entries of the virtual queue are stored on *QueueChunks*, referenced in the chunk pointer queue. A chunk size of 8 KiB reduces the static size to 1/2048 of the original queue plus one allocated *QueueChunk* to initialize the queue. For enqueue and dequeue operations on an *VAQ*, the access management has to check for availability of the referenced queue slot. Each thread still determines queue positions using `atomics` on the

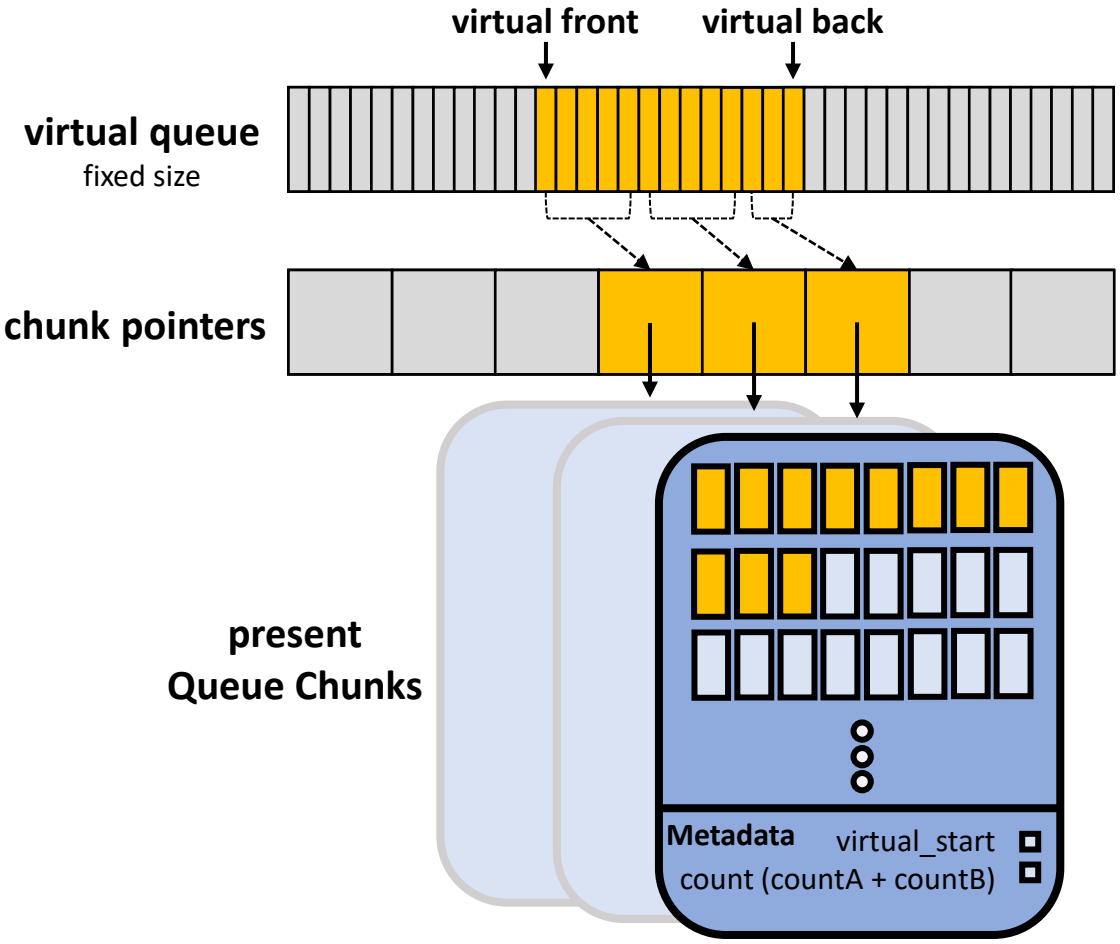


Figure 5.3: The larger, static queue is replaced by a much smaller chunk pointer queue, where each element points to a *QueueChunk* holding the actual queue data. Each *QueueChunk* holds information regarding the amount of queue storage that is currently present as well as a *virtual* starting index, referring to the virtual position in the queue.

`front` and `back` index. However, these positions are now *virtual* positions in the queue. The *QueueChunk*, which holds the real position, is determined by dividing the virtual position by the number of items per *QueueChunk* modulo the chunk queue size. As this *QueueChunk* might not have been placed in the chunk pointer queue yet, threads have to check if this *QueueChunk* is already present. This check follows the same procedure as already introduced in Section 4.2.2 to wait for the index to become valid.

The thread assigned to position 0 on a *QueueChunk* pre-emptively allocates a new *QueueChunk* from the memory manager, initializes it and places it in the next slot of the chunk pointer queue—we also place one chunk during initialization. The placement during enqueue is carried out before accessing the queue element at position 0 to avoid

serialization of chunk pointer enqueues and to reduce waiting time for other threads that want to access that chunk. Since the top-level chunk pointer array is always present, different allocating threads don't have to wait for their own chunk to exist before they can allocate a new *QueueChunk*. Interleaving allocations like this further reduces the waiting time for other threads. Threads waiting on a *QueueChunk* employ a strategy of exponential back-off. While threads are waiting for a *QueueChunk*, i.e., if the chunk pointer is a *deletion marker*, they back-off with a progressively larger timeout value for each failed check (only possible on $CC \geq 7.0$, below that calls to `sleep()` are replaced by calls to `threadfence()`). Once the correct *QueueChunk* has been located, the low-level enqueue and dequeue operations follow the same principle as detailed in Algorithm 5.

Each *QueueChunk* has two counters to determine the current fill-level (`countA` and `countB` in Figure 5.3), which are stored in a single variable to allow for simultaneous `atomic` manipulation. After an *enqueue*, both counters are incremented; a *dequeue* decrements `countB`. If a *QueueChunk* has been fully used and emptied, `countA = #spots` and `countB = 0`. Since the dequeues replaced all elements with *deletion markers*, an emptied *QueueChunk* can immediately be returned for re-use to the respective queue in the memory manager. The major difference to the previous approaches is the top-level *QueueChunk* management. While this leads to one indirection and waiting overhead depending on the queue access pressure, the *VAQ* greatly reduces static storage requirements.

5.4.2 Virtualized Linked-Chunk Queue (*VLQ*)

The *VLQ* (Figure 5.4) replaces the already smaller chunk pointer queue of *VAQ* with a *linked chunk pointer queue*, reducing the static storage requirements to just three pointers (`front`, `back` and `old`). Removing the static queue also removes the static size limit, as queues can grow arbitrarily large and shrink to virtually nothing (once again, at least one *QueueChunk* remains). Similar to the *VAQ*, threads determine their virtual enqueue and dequeue position atomically but now start traversal at either `front` or `back`. Each *QueueChunk* stores the `virtual_position` of its first slot, such that threads can locate their *QueueChunks* and stop traversal.

During *enqueue*, a thread reads the current `back` and uses the `virtual_position` to determine if it is at the correct *QueueChunk* and if so, performs the *enqueue*. Otherwise, it traverses to the next *QueueChunk* and so on. If the next *QueueChunk* has not been placed in the list yet, it spins on the next pointer using exponential back-off, until it is available and the thread can continue the traversal. The thread with position 0 on a *QueueChunk* again pre-emptively allocates the next chunk as in *VAQ*. Note that we allocate multiple *QueueChunks* in parallel before they are placed, as threads can determine whether they are assigned to a first slot on a chunk from `virtual_position`. Only the placement itself, i.e.,, setting the next pointer on the previous *QueueChunk*, is inherently serial.

If `countA = #spots` after an enqueue, *all* enqueues on this chunk have been finished and the *back* pointer can be moved using Algorithm 10. Due to the potentially high

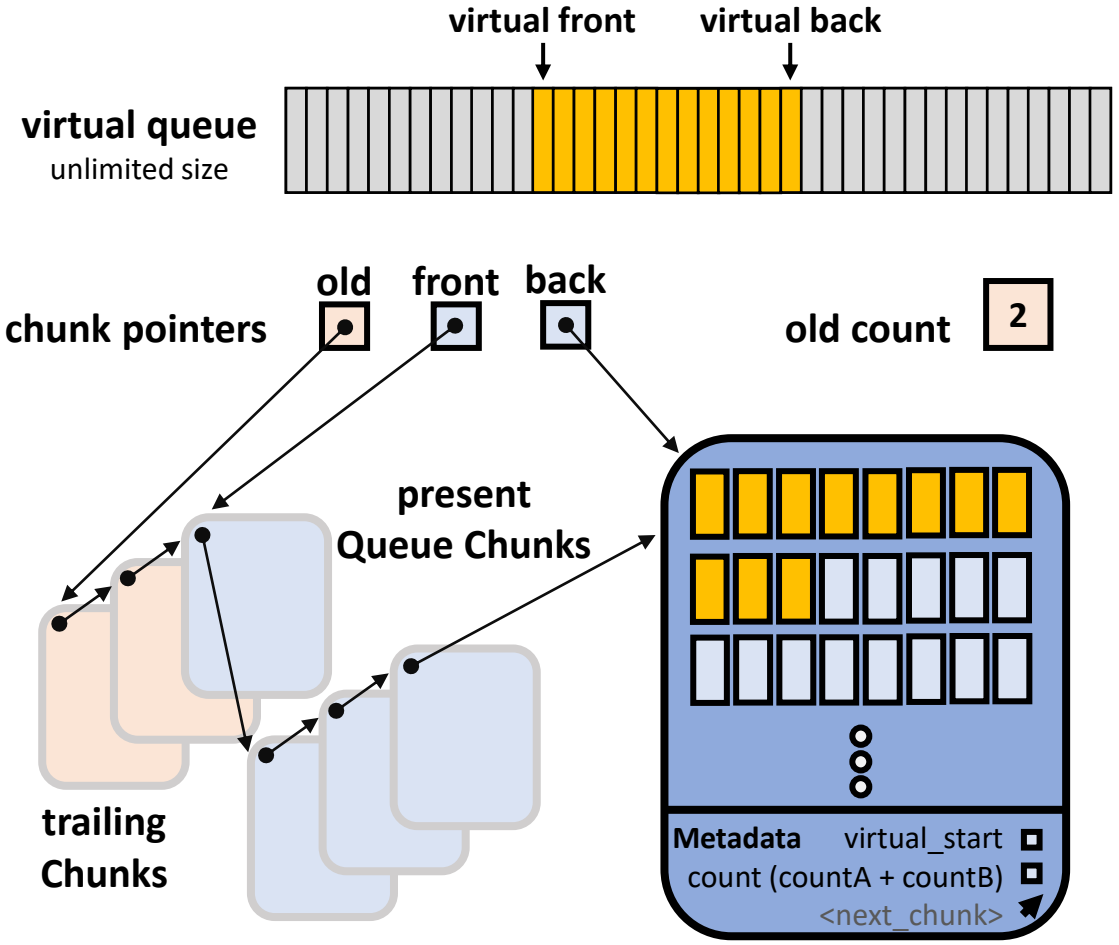


Figure 5.4: Compared to *VAQ*, the *chunk pointer queue* is replaced with just three pointers, pointing to the front and back (as well as trailing) *QueueChunks*. Each *QueueChunk* also has a pointer to the next *QueueChunk*, additional to the two counts and virtual starting index.

Algorithm 10: Move front/back pointer along

```
1 Function Chunk::setPointer(ptr)
2     chunk ← this
3     while atomicCAS(ptr, chunk, chunk.next) = chunk do
4         if chunk.next.countA = #spots then
5             chunk ← chunk.next
```

pressure on the queue, threads are not guaranteed to see their chunks being full in the correct list order. Hence, only the thread that fills up the *QueueChunk* to which `back`

currently points, moves it along the list to the first non-full chunk. After a successful swap, the thread continues with the next chunk as it may have filled up earlier (and the `atomic-Compare-and-Swap` of another has failed). After a successful swap operation, the next chunk is checked as well for `countA = num_spots` (see line 4) and the operation is repeated as long as the next chunk is full. If a swap operation does not succeed, no further changes are required, as some other thread will perform it later. This reduces the amount of traversal, as only threads, which read the `back` pointer before the update have to traverse. As each *QueueChunk* holds many queue slots (2048 on 8 KiB chunks), the traversal is further reduced.

A *dequeue* operation starts from `front` and traverses to its assigned *QueueChunk* using the virtual position. The low-level dequeue is performed as in the *VAQ*; after a successful dequeue, `countB` is decremented. If `countA = #spots` and `countB = 0`, all enqueue/dequeue operations on a chunk are completed and the front pointer is moved, as in Algorithm 10. Additionally, we count the successful moves.

Moving the `front` pointer does not immediately remove the corresponding *QueueChunks*. This is crucial, as other threads may still be reading from this *QueueChunk* during traversal, hence immediate re-use might overwrite data still in use. Hence, removing these *QueueChunks* too early would potentially prohibit forward progress for certain threads. While we could use hazard pointers [34], they would introduce a significant overhead. Instead, we delay the clean-up by introducing the `old` pointer, which lags behind the actual `front`. When moving `front`, we increment another variable, `old_count`, and only if it passes a threshold `t`, `old` is also moved and the *QueueChunks* are submitted for reuse. Thus, we always leave a trail of $\geq t$ chunks behind `front`. `t` is determined heuristically from the number of potentially concurrently-active threads, which is limited for each GPU. This results in an estimate of how many threads might actually depend on an old, invalid `front` pointer. The `old` pointer is moved as `front`, but the *QueueChunks* are submitted for re-use.

For all operations (*enqueue*, *enqueueChunk* and *dequeue*), given the on-the-fly linked-list structure, we have to consider some GPU architecture pitfalls as well. Prior to $CC < 7.0$, warps always execute in lock-step, individual threads within a warp might take different branches, but this just masks the other threads and all threads within a branch once again execute together. To take an example, if one thread within a warp is responsible for allocating the next *QueueChunk* but other threads within the same warp depend on this *QueueChunk*, it is important to force the hardware to first allocate the next chunk before letting the other threads execute. This can be accomplished by building on warp-synchronization primitives to let threads within a warp determine what kinds of jobs each one has to do. At the beginning, all active threads are queried and all check, if at least one of them has to perform work on the current chunk. If this is true at least for one of them, these threads are prioritized to do their work first, halting the other threads, as can be seen in Algorithm 11 in lines 5 to 9.

Algorithm 11: Guarantee warp progress by letting all threads successively over chunks

```
1  Function guaranteeWarpProgress(position)
2      activemask ← __activemask()
3      work_not_done ← true
4      while true do
5          predicate ← checkVirtualStart(position)
6          if __any_sync(activemask, predicate) then
7              if predicate then
                   // Execute enqueue/dequeue
8                  work_not_done ← false
9          __sync_warp(activemask)
10         if __any_sync(activemask, work_not_done) then
11             if work_not_done then
                   // Traverse to next chunk
```

5.5 Framework

As part of a wider survey, we provide our complete test framework as well as an interface to all tested memory managers in a public repository on GitHub. This includes *CUDA-Allocator*, *XMalloc*, *ScatterAlloc*, *Halloc*, *RegEff* and *Ouroboros*. *FDGMalloc* is also included, but crashes in most test scenarios and also does not allow for general purpose free and re-use over multiple kernel launches, hence it was omitted for the final evaluation. All frameworks, except for *Ouroboros* and the *CUDA-Allocator*, are configured to generate code for the *pre-Volta* architecture, as they rely on warp-synchronous behavior to function correctly.

Algorithm 12: End-to-end usage example

```
   // MemoryManagerType can be any of the listed memory managers
1  using MM = MemoryManagerType;

   // GPU code
2  __global__ void deviceKernel(MM mm)
3      void* ptr = mm.malloc(size);
4      mm.free(ptr);

   // CPU code
5  MM mm(SIZE);
6  deviceKernel<<<gridSize, blockSize>>>(mm);
```

Algorithm 12 shows a typical usage example. Each memory manager is instantiated

on the host with a configurable size of the manageable memory. This memory manager can then be passed to device kernels and offers the standard *malloc/free* interface. Using this framework, one can integrate a memory manager into an existing project and simply swap out one declaration to change between memory managers, allowing for a simple benchmarking setup. The full test suite (including results for the NVIDIA TITAN V and NVIDIA RTX 2080Ti) can be found on GitHub.

5.6 Evaluation

All performance measurements were conducted on an NVIDIA TITAN V (12 GB V-RAM) and an Intel Core i7-7700 with 32 GB of RAM and took around 600 h (roughly $3\frac{1}{2}$ weeks) to complete. Additional results on an NVIDIA RTX 2080Ti (11 GB V-RAM) can be found on GitHub. The framework is `CMake`-based and runs both on Linux and Windows. All given results were captured on Linux with `gcc` 10.2.0 using NVIDIA CUDA 10.2. Not all tested frameworks also work correctly with independent thread scheduling behavior introduced with the Volta generation of NVIDIA cards [41]. For these, we pass `compute_60` to the compiler to enforce warp-synchronous execution. Considering *Ouroboros*, $S|VA|VL$ denote standard, virtualized array-hierarchy and virtualized linked-chunk methods, $P|C$ define if page or chunk indices are stored. This results in the variants *Ouro-S-P*, *Ouro-S-C*, *Ouro-VA-P*, *Ouro-VA-C*, *Ouro-VL-P* and *Ouro-VL-C*.

All frameworks were setup with 8 GB of manageable memory. Only the *out-of-memory* testcase was initialized with 2 GB for reduced run times. Variants of *RegEff* were built with *warp-coalescing* turned off, as this did not work for any of the testcases. We use as a baseline a simple memory manager built on `atomics` on a shared `offset` (referred to as *Atomic*), but this is no true memory manager due to the lack of deallocation. We use a consistent color scheme throughout all plots to save on space, this color map can be seen in Figure 5.5.

5.6.1 Initialization & Register Requirements

Evaluating initialization performance, the *CUDA-Allocator* only sets its size limit and hence is clearly fastest (≤ 0.05 ms), followed by *Atomic* and standard variants of *Ouroboros* ($\sim$6 ms). The virtualized variants of *Ouroboros* take a little bit longer than the standard variant as the memory has to be flushed with the *deletionmarker* first, so that each chunk is instantaneously usable for any kind of purpose. All other approaches are close in initialization performance (30 ms–40 ms), except for *Halloc*, which is on about 5.5× slower

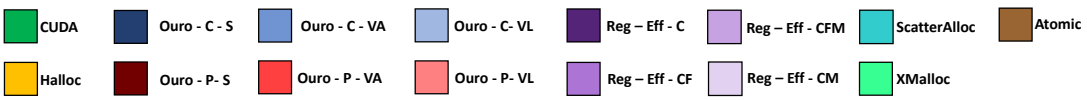
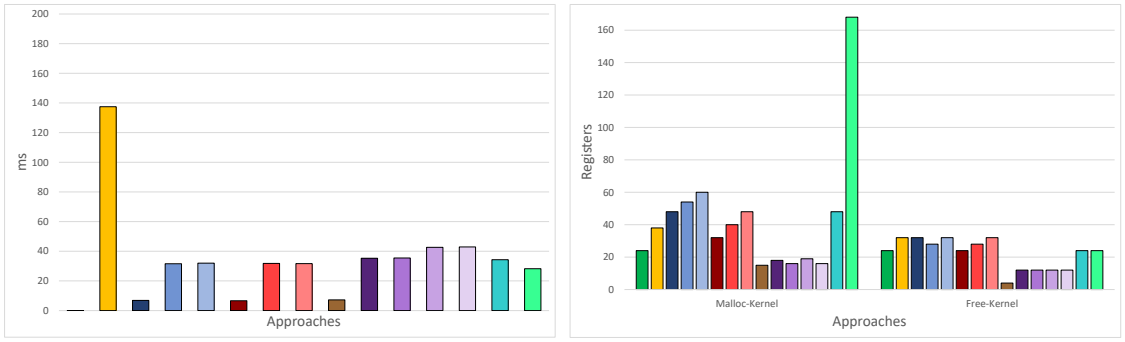


Figure 5.5: Color scheme used henceforth for all tested approaches.



(a) Initialization Timing (b) Register Requirements

Figure 5.6: Manageable Memory Initialization Timings as well as register requirements for `malloc()` and `free()` respectively.

compared to the average initialization time, as can be seen in Figure 5.6a.

We also evaluate register requirements for *malloc* and *free* respectively, as can be seen in Figure 5.6b. The respective *malloc* implementation requires more registers than *free* for all approaches. The four variants of *RegEff*, as suggested by the paper title, use the least amount of registers both for *malloc* and *free*, closely followed by the *CUDA-Allocator*. *Halloc* and *ScatterAlloc* require around 40 registers for *malloc* and between 20-30 registers for a call to *free*. *Ouroboros* is slightly more resource intensive for the *malloc* case, with around 50 registers for the chunk-based approaches and around 40 registers for the page-based counterparts, while *free* is similar to *Halloc* and *ScatterAlloc* with slightly more than 20 registers. Only *XMalloc* shows a large discrepancy between *malloc* (168) and *free* (24).

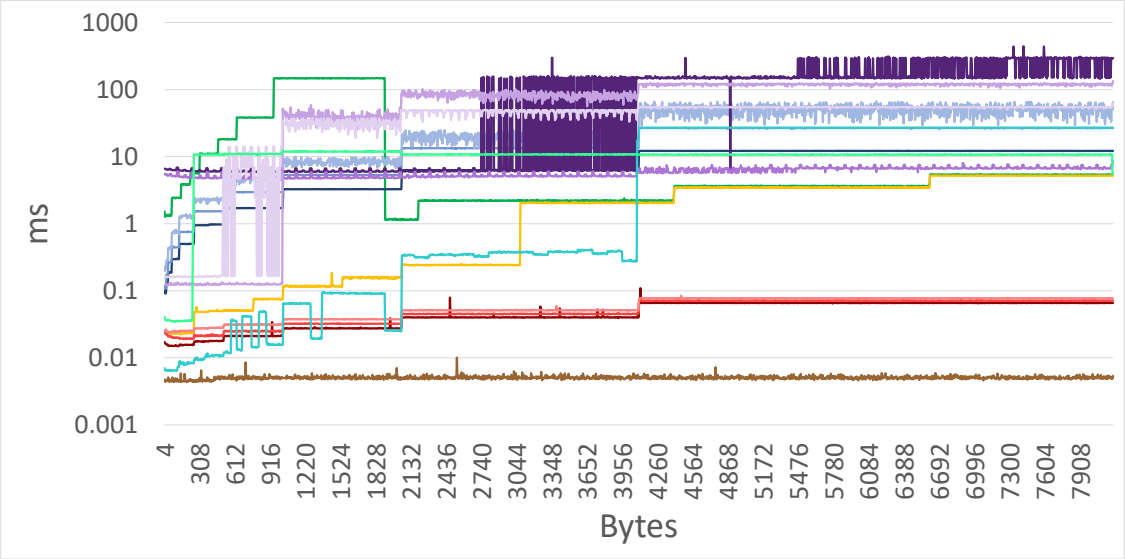
5.6.2 Allocation Performance

To evaluate allocation performance, we investigate three different scenarios, all tested on the range 4 B–8192 B:

- Allocation performance for 10 000 and 100 000 allocating threads or warps
- Allocation performance for mixed sizes (thread-based)
- Performance scaling for varying numbers of threads for powers-of-two between 2^0 - 2^{20}

5.6.2.1 Allocation Performance for Allocation Size

We test 10.000 (as shown in Figure 5.7) and 100.000 (as shown in Figure 5.8 and Figure 5.9) allocations in the range between 4 B–8192 B. For comparison with the *NVIDIA Titan V*, performance on the *NVIDIA RTX 2080Ti* is shown in Figure 5.10. As the overall pattern remains remarkably the same, we will showcase results only from the TITAN V for sake of brevity (full results can be found on GitHub).



(a) Allocation performance 10K, with *ScatterAlloc* performing best for small allocations and *page-based Ouroboros* being the overall best choice.

(b) Deallocation performance 10K, best performers are variants of *RegEff* and *ScatterAlloc*.

Figure 5.7: Thread-based allocation/deallocation performance for 10 000 allocations (5.7a and 5.7b) for the range 4 B–8192 B.

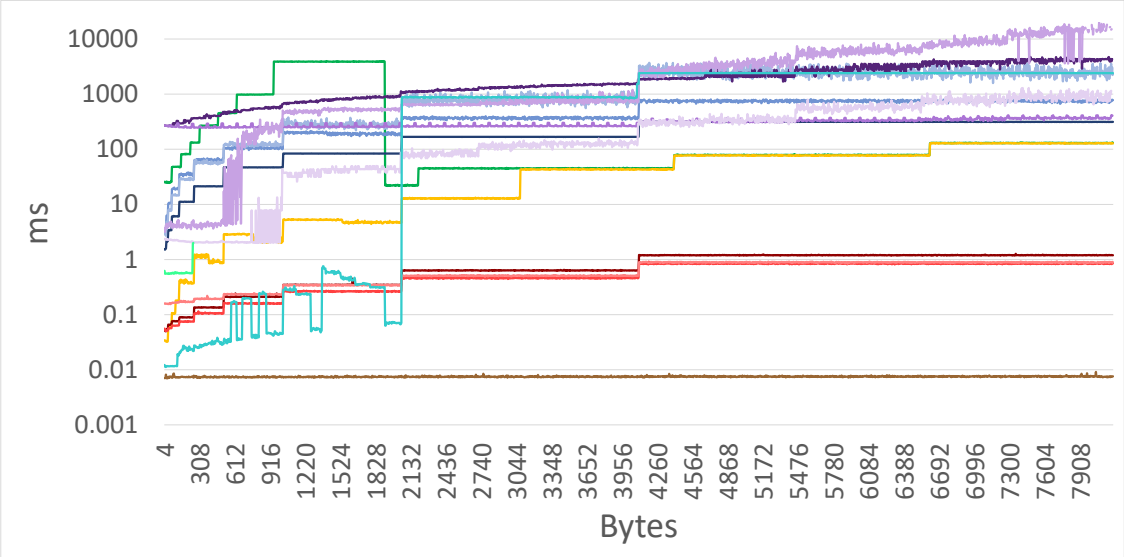


Figure 5.8: [**Titan V**] Thread-based allocation performance 100K, small allocations favoring *ScatterAlloc* with *page-based Ouroboros* performing best overall again.

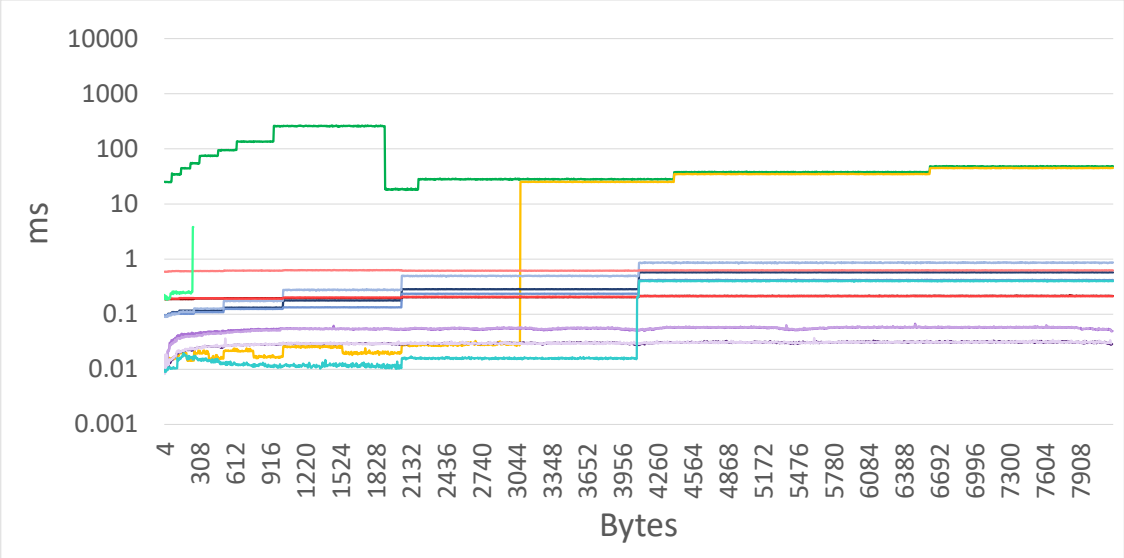
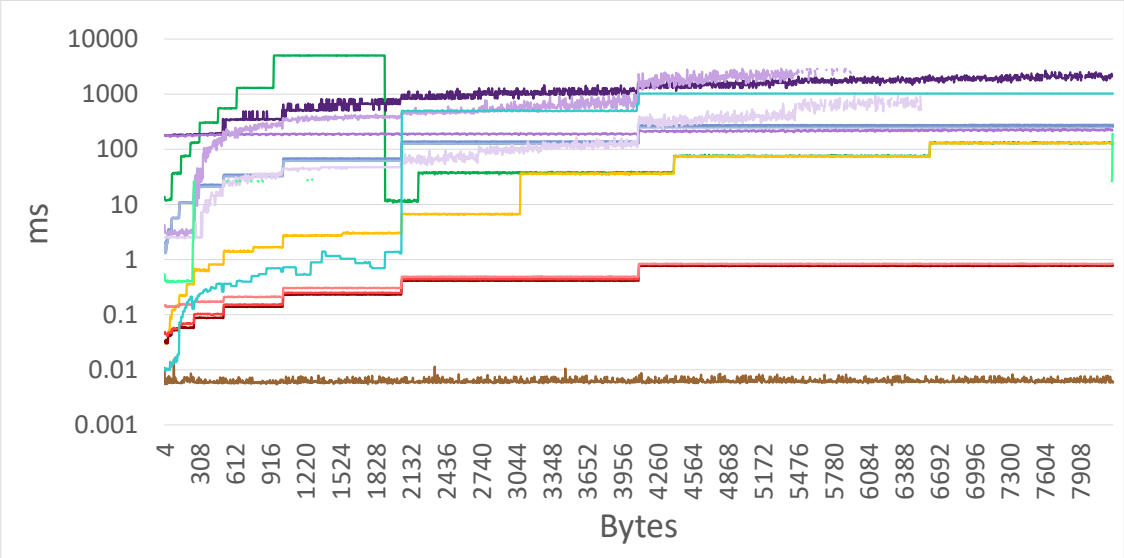
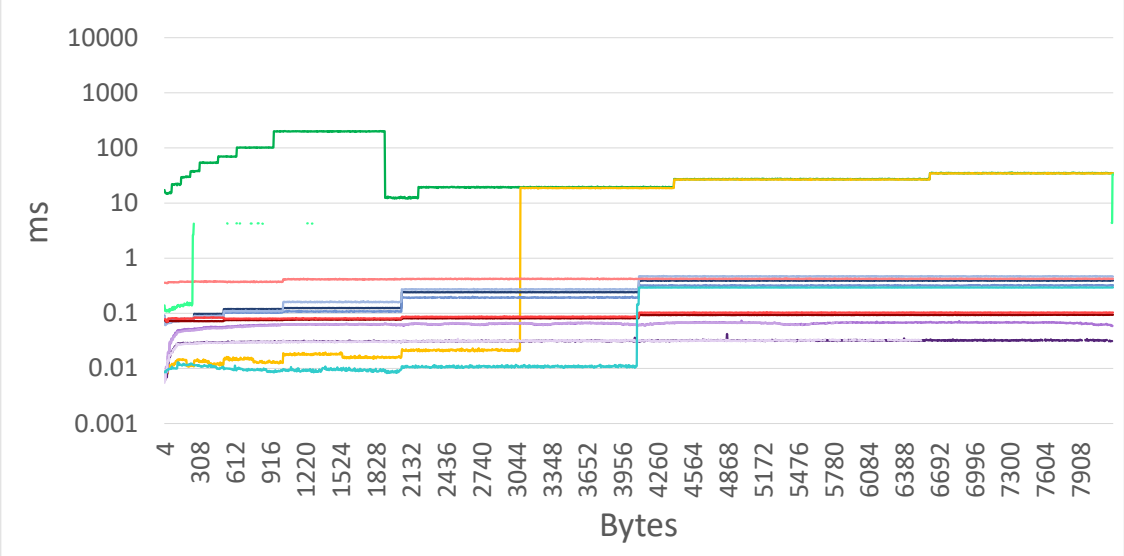


Figure 5.9: [**Titan V**] Thread-based deallocation performance 100K, best performance shown by *ScatterAlloc*, *Halloc* and variants of *RegEff*.

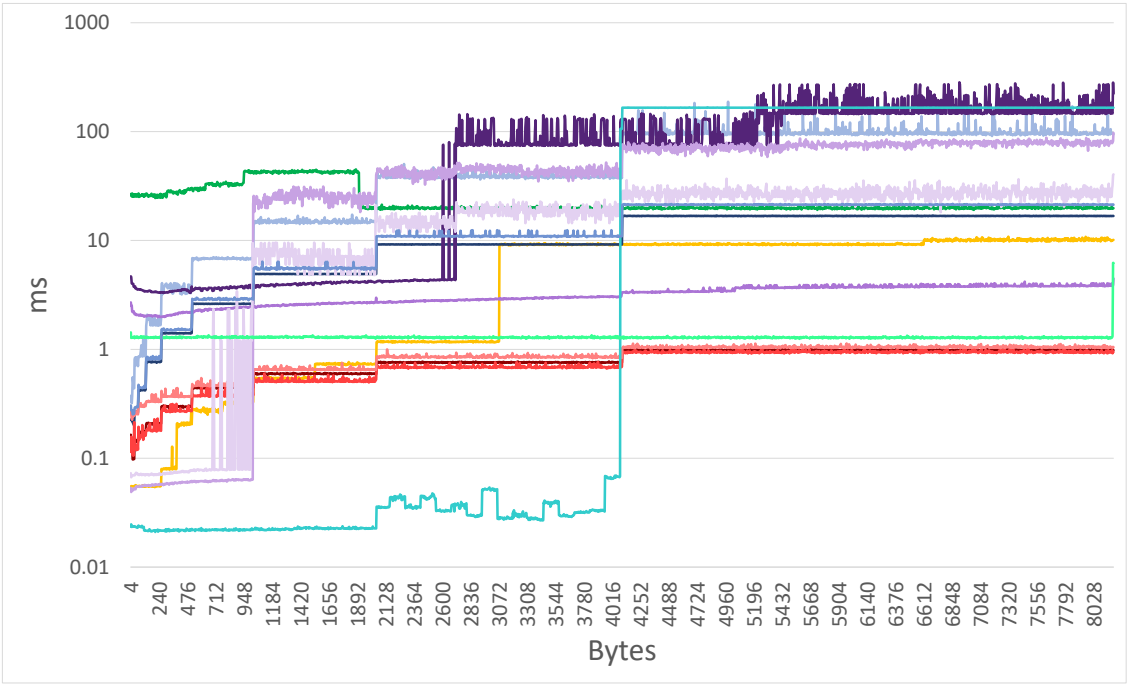


(a) **[2080Ti]** Allocation performance 100K, here *page-based Ouroboros* is the best choice even for smaller allocations as well as for large onces.

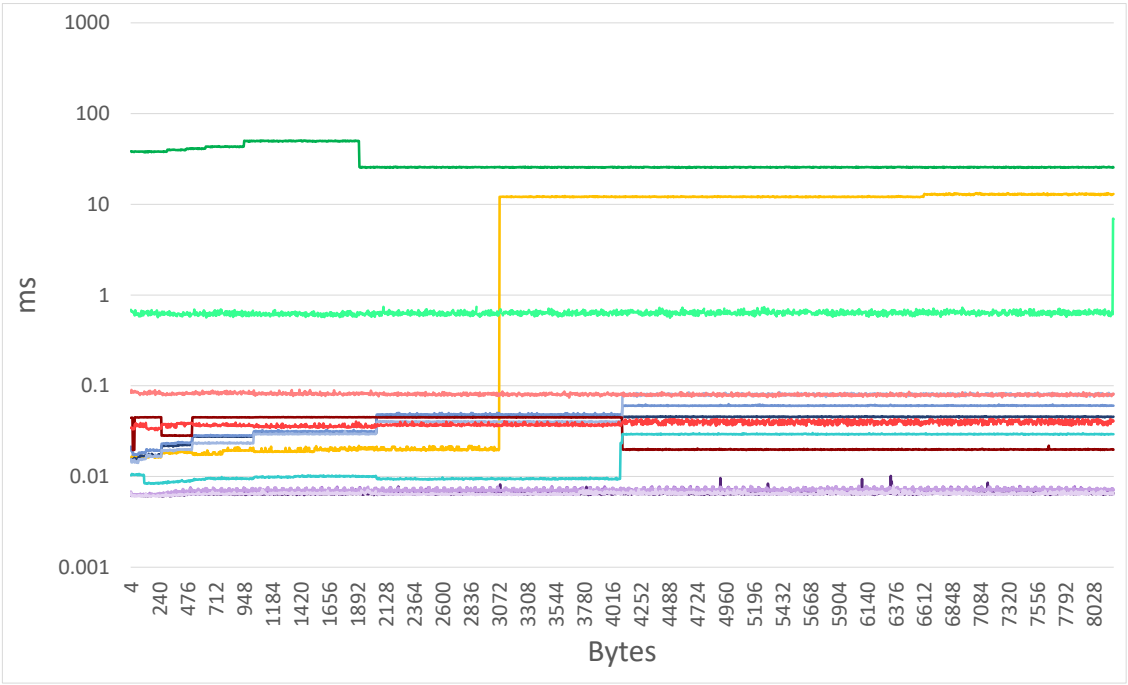


(b) **[2080Ti]** Deallocation performance 100K, best performance achieved by *ScatterAlloc*, *Halloc* and variants of *RegEff*.

Figure 5.10: **Thread-based** allocation/deallocation performance for 100.000 allocations (**5.10a** and **5.10b** on the NVIDIA RTX 2080Ti) for the range 4 B–8192 B, which shows the same overall pattern as Figure 5.8 and Figure 5.9.



(a) Allocation performance 10K, with *ScatterAlloc* improving its lead for small allocations and *Halloc* also improving for small allocations.



(b) Deallocation performance 10K, still *ScatterAlloc* and variants of *RegEff* perform best here.

Figure 5.11: **Warp-based** allocation and deallocation performance for 10 000 allocating warps (one thread per warp allocating).

Warp-based allocation and deallocation performance (with one thread per warp allocating) with 10.000 warps can be seen in Figure 5.11. The performance results suggest that the *CUDA-Allocator* has some larger, divisible unit that can be split into smaller sizes. This is clearly visible in the characteristic staircase pattern visible for both allocations and deallocations. Interestingly, performance of the native CUDA allocator can be divided into three intervals:

- Allocations $\leq$ 64B: the time per allocation is constant, indicating a padding to 64 B;
- Allocations between 64 B–1024 B: the allocation time increases with size;
- Allocations larger than 2048 B: allocation is approximately 110–160$\times$ faster than a 1024 B allocation, followed again by an increase in time.

Furthermore, it seems it has more than one size for this unit, as there is a clear split in performance right before 2048 B. *CUDA-Allocator* also is the only approach with deallocation performance consistently above 1 ms. *CUDA-Allocator* is seemingly not directly re-using freed pages, as there is no difference between the first and any following allocation after freeing the previously allocated memory.

ScatterAlloc performs best (staying even close to the *atomic* baseline) until it has to start searching for contiguous free blocks, resulting in a steep drop in performance at around 2048 B. Performance for *Ouroboros* is double-edged. The *chunk-based* variants are considerably slower than *ScatterAlloc*, but outperform the *CUDA-Allocator* up to its unit split. The *page-based* variants are very close in performance to *ScatterAlloc* for smaller sizes, but considerably outperform all other approaches for larger sizes. *Halloc* performs well until its hand-off to the *CUDA-Allocator*. *RegEff* does not perform well with *thread-based* allocation methods. This is not helped by the problem that *warp-coalescing* (which would allocate one large allocation for all allocation requests within a warp) does not complete in any of the testcases, as there seem to be problems with deleting parts of this larger allocation. *XMalloc* falls in between *CUDA-Allocator* and *Halloc* performance-wise, but is unstable, only being able to finish the testcase for 10.000.

Considering deallocation performance, the results once again show a clear split between the *CUDA-Allocator* and all other approaches. Interestingly enough, performance still follows a staircase pattern and is orders of magnitude slower. *XMalloc* falls again in the middle but remains unstable at higher thread counts. The variants of *Ouroboros* are next, as they not only alter some state information during free, but also potentially enqueue pages or chunks into the respective queues. Slightly faster still are *ScatterAlloc*, *Halloc* and the variants of *RegEff*, which only modify state information and do not have to maintain an additional data structure. Nonetheless, it is important to note that considering overall performance, allocation performance clearly dominates as it is typically orders of magnitude slower than deallocation.

Warp-based allocation changes the picture somewhat, as can be seen in Figure 5.11, in that *Ouroboros* slows down a little, while *RegEff* gains some performance. Interestingly,

the *CUDA-Allocator* also sees a change in performance, but mainly reducing the range of performance fluctuation. *Halloc* now outperforms *page-based Ouroboros* for allocations $\leq$ 1024 B and the two *Multi-RegEff* variants also start strong, but have an issue with repeated allocations/deallocations, slowing down significantly over time. Overall, the choice still remains between *ScatterAlloc* and *page-based Ouroboros*.

5.6.2.2 Mixed Allocation Performance

This testcase tries to highlight performance numbers during mixed allocation, i.e., if different allocation sizes are allocated during one kernel call. To evaluate this, each thread requests an allocation from a certain range of available sizes. The lower bound is 4 B, while the upper bound ranges between 4–8192 B, a value is randomly chosen in this range. Once again, we look at 10.000 as well as 100.000 allocating threads, allocation performance for 100.000 is shown in Figure 5.12.

Considering smaller allocation ranges, *ScatterAlloc* clearly performs best once again, followed by *Halloc* and *page-based Ouroboros*. After increasing the range to 4–1024 B, *page-based Ouroboros* clearly shows its strength. The *CUDA-Allocator* shows its characteristic spike at 2048 B, after which performance increases again. Clearly visible are also the struggles experienced by the variants of *RegEff*, which frequently crashes in this kind of environment. *XMalloc* is very stable over the range where it still works, but larger allocation sizes still pose problems.

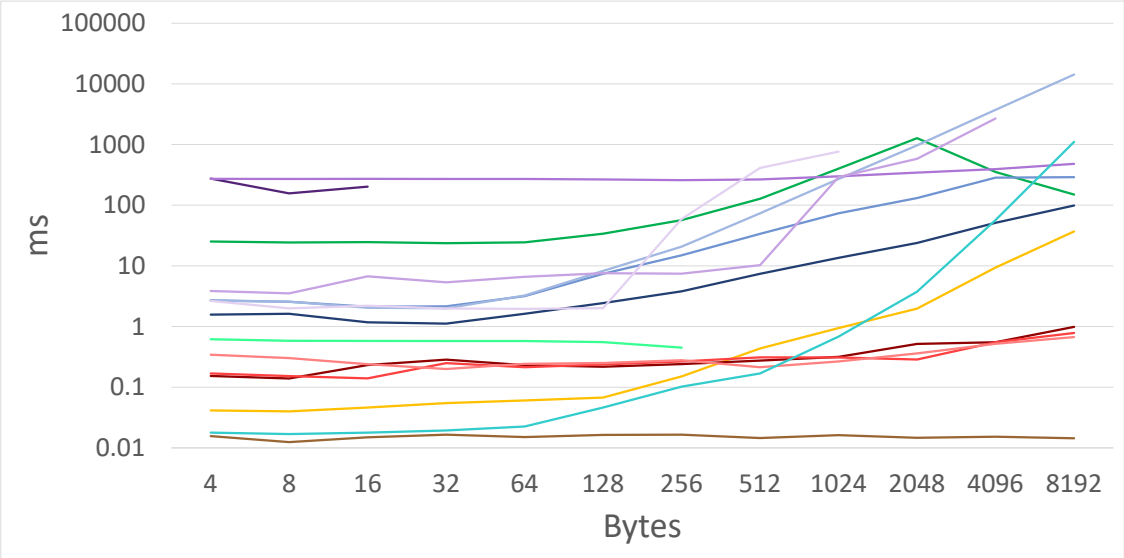
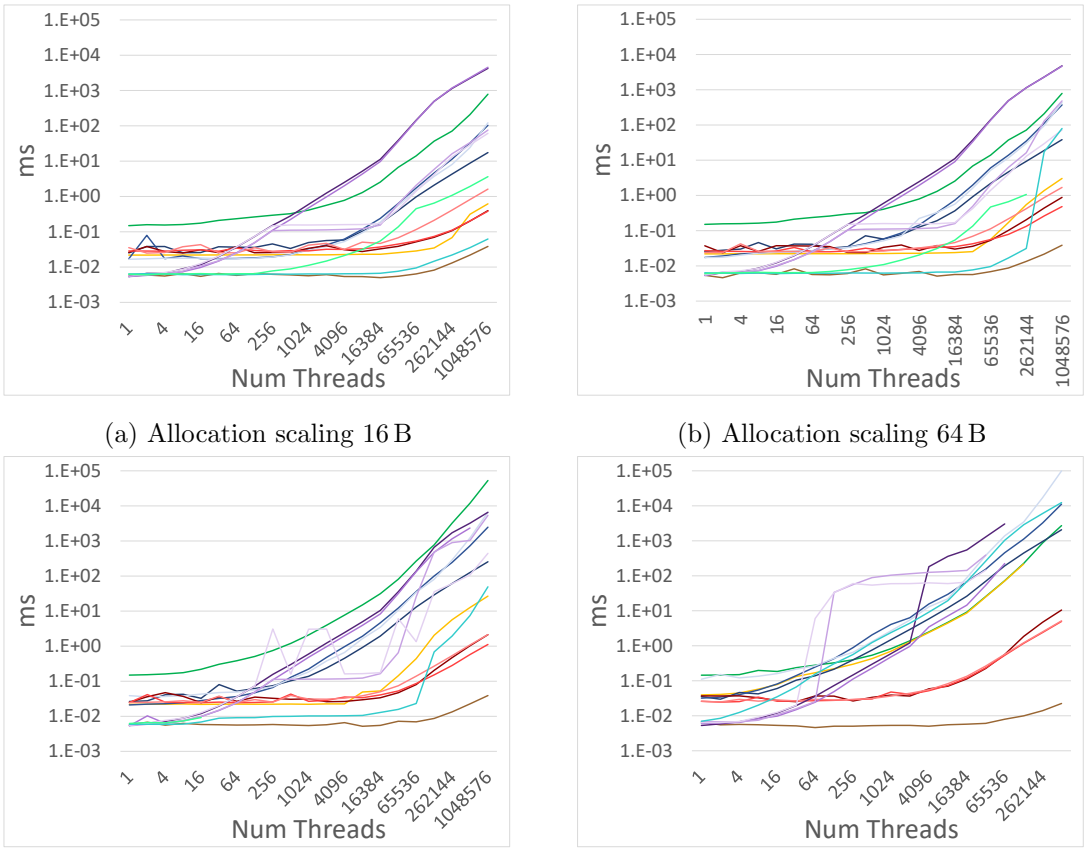


Figure 5.12: Mixed allocation performance for 100.000 allocations in the range 4 B to 4 B–8192 B (4 B–4 B, 4 B–8 B, $\cdots$, 4 B–8192 B)



(a) Allocation scaling 16 B

(b) Allocation scaling 64 B

(c) Allocation scaling 512 B

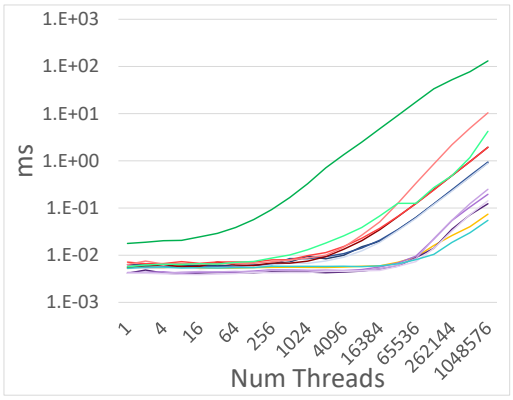
(d) Allocation scaling 8 KiB

Figure 5.13: Allocation performance scaling for 16 B, 64 B, 512 B and 8 KiB (**5.13a** - **5.13d**).

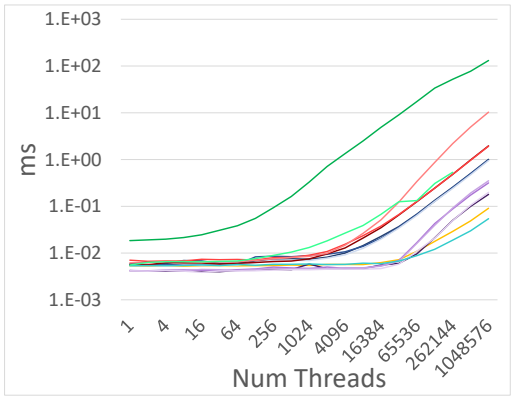
5.6.2.3 Performance Scaling

To assess performance scaling, we test the range of 4 B–8192 B and vary the number of threads between 2^0 - 2^{20}. Four examples are shown in Figure 5.13a to Figure 5.13d.

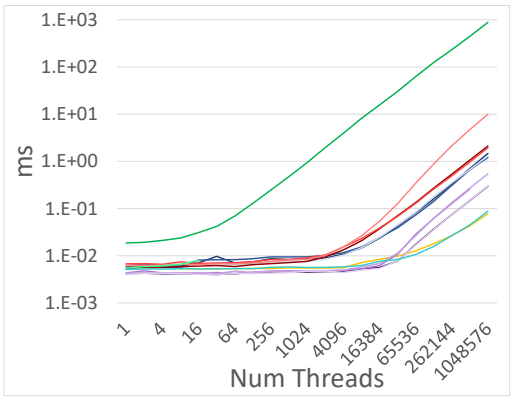
The *CUDA-Allocator* shows a similar pattern over the whole range, staying relatively flat up until 1000 threads and then slowly increase for increasing numbers of threads. Especially for smaller allocations sizes, its comparatively poor performance is clearly noticeable, only surpassed by certain variants of *RegEff*. Figure 5.13a and Figure 5.13b still show *Halloc* keeping up well with *ScatterAlloc*, which performs clearly best and stays even close to the atomic baseline, and *page-based Ouroboros*, which has a very consistent slope over all tested sizes. All three remain flat for one order of magnitude longer than the other approaches. But especially for increasing allocation sizes, as can be seen in Figure 5.13c and Figure 5.13d, *Halloc* first and then also *ScatterAlloc* slows down, while *page-based Ouroboros* still shows a very consistent performance profile over the full range. Variants of *RegEff* show an unusual pattern as they start decreasing in performance much earlier



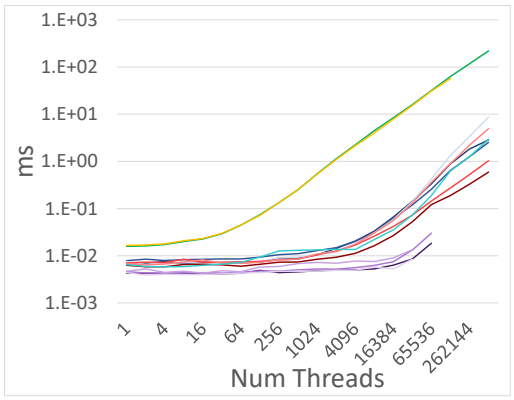
(a) Deallocation scaling 16 B



(b) Deallocation scaling 64 B



(c) Deallocation scaling 512 B



(d) Deallocation scaling 8 KiB

Figure 5.14: Deallocation performance for 16 B, 64 B, 512 B and 8 KiB (**5.14a** - **5.14d**).

compared to the other approaches. This is true even for small thread counts, as can be seen in Figure 5.13a and Figure 5.13b. Figure 5.13d shows the performance discrepancy between *page-based Ouroboros* and all other variants very clearly for larger allocation sizes. *XMalloc* shows good performance, especially for very small numbers of threads and also smaller allocation sizes, but is too unstable for larger number of threads or allocations.

Considering deallocation performance in Figure 5.14a to Figure 5.14d, performance is much more homogeneous, as there is little difference between different allocation sizes. The *CUDA-Allocator* once again is left behind and for smaller allocation sizes, there exist a small performance gap between *Ouroboros* and the other approaches, which closes for larger allocation sizes. This gap can be explained by the additional work to be done by the *queue*-based *Ouroboros*, which, compared to the others, not only updates state variables but also inserts elements into a queue.

5.6.3 Fragmentation

We consider two testcases to evaluate fragmentation. We explore fragmentation during allocations of different sizes and efficient memory usage with an out-of-memory testcase.

5.6.3.1 Fragmentation Range Testcase

To assess fragmentation from outside the allocators, we track the maximum address range for a number of allocations as well as the maximum address range after 100 iterations of allocations and deallocations. The former result can be seen in Figure 5.15. *CUDA-Allocator* always reports back the maximum possible range, which might suggest that it starts allocating from both ends of its memory region. The same is true for *XMalloc* (which crashes early unfortunately). *Ouroboros* stays close to the baseline and shows the best utilization, given its alignment to powers of two. *Halloc* comes second, followed by *ScatterAlloc* and then *RegEff*.

5.6.3.2 Out-Of-Memory Testcase

This testcase performs allocations until either out-of-memory is reported by the system or an allocator did not finish within an hour of runtime. Figure 5.16 reports how often such an allocation with 100.000 threads was possible as a ratio of the maximum number of iterations possible given the memory size. The alignment of 16 B is clearly visible here, as all approaches report increased utilization between 4 B up to 16 B. *Ouroboros* clearly shows the best utilization, with 98 % or higher for all variants after 16 B. *ScatterAlloc*

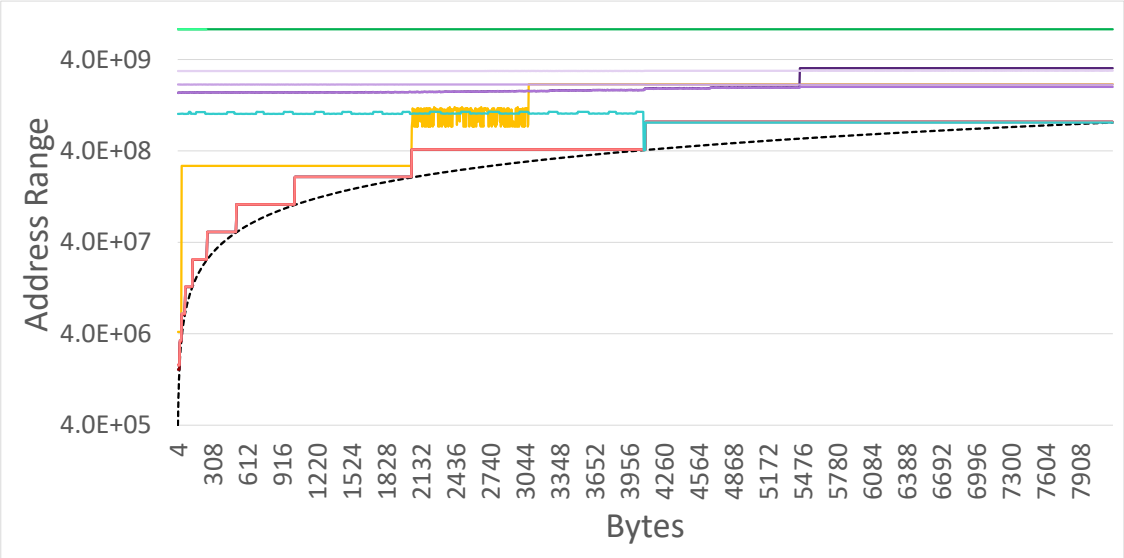


Figure 5.15: Fragmentation with theoretical baseline, tracking the maximum address range returned by the allocators for given allocation size with 100.000 allocations.

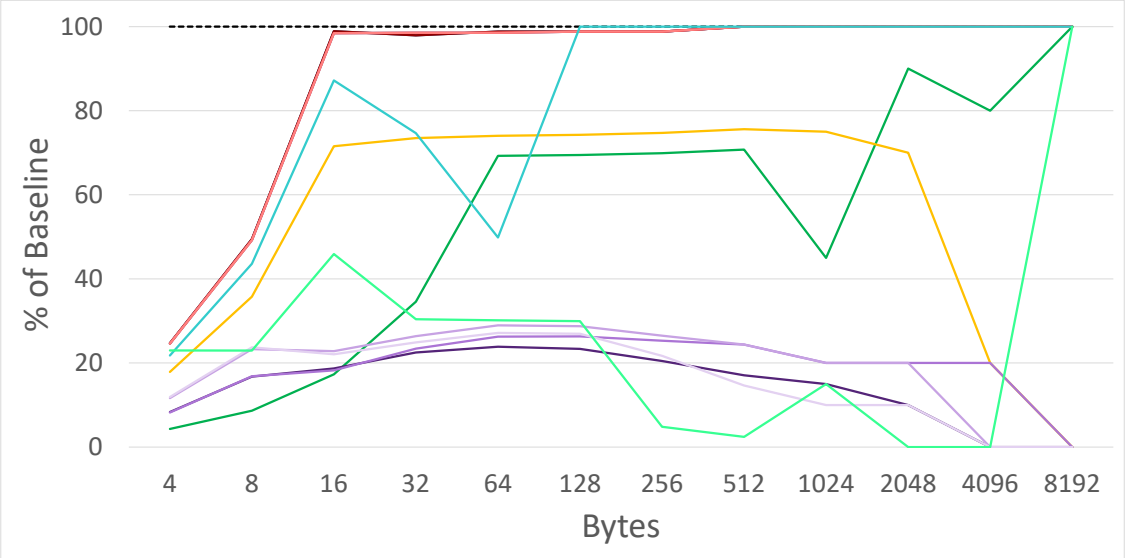


Figure 5.16: Out-of-Memory testcase, tracking the number of allocation rounds per size as a fraction of the theoretical baseline.

comes second, even reaching full utilization halfway through the test case. *Halloc* cannot reach full potential for larger sizes, as these are directed towards the *CUDA-Allocator*.

For smaller sizes, it comes close to 75 %, which is due to marking chunks as busy early and the reduced memory size due to the split with the *CUDA-Allocator*. *CUDA-Allocator* as well as *RegEff* do not finish but are reined in by the one hour mark (the other approaches typically finish each test case in less than a minute), as both approaches slow down with an increasing number of allocations. *XMalloc* has problems with stability, returning various violations if memory is not freed.

5.6.4 Real-World Performance

5.6.4.1 Work Generation

This test case emulates a real-world example of a set of threads producing work. The memory manager performance can then be compared to the canonical approach of using a prefix-sum plus allocation from the host.

We test two ranges, 4–64 B (in Figure 5.17) of work generated per thread as well as 4 B–4096 B (in Figure 5.18). We launch an increasing number of threads and also compare to the *Baseline* built on a prefix-sum from *CUB*. For the smaller range, as in Figure 5.17, only *ScatterAlloc* is able to consistently outperform the *Baseline*, *Halloc* also stays very close over this range. *Page-based Ouroboros* shows similar performance up to a few thousand threads and then falls slightly behind, with all other approaches considerably slower. For the larger range, as in Figure 5.18, *Halloc* slows down, with only *ScatterAlloc* and *page-based Ouroboros* outperforming the *Baseline* up to tens of thousands of threads.

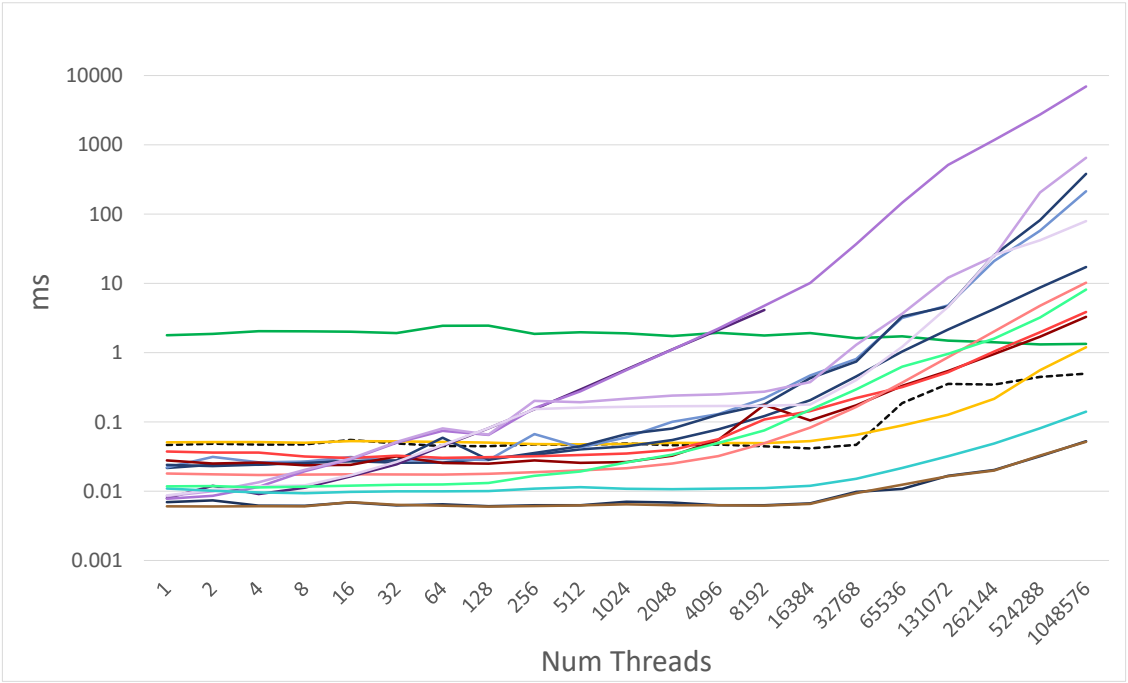


Figure 5.17: Work generation (4 B–64 B per thread)

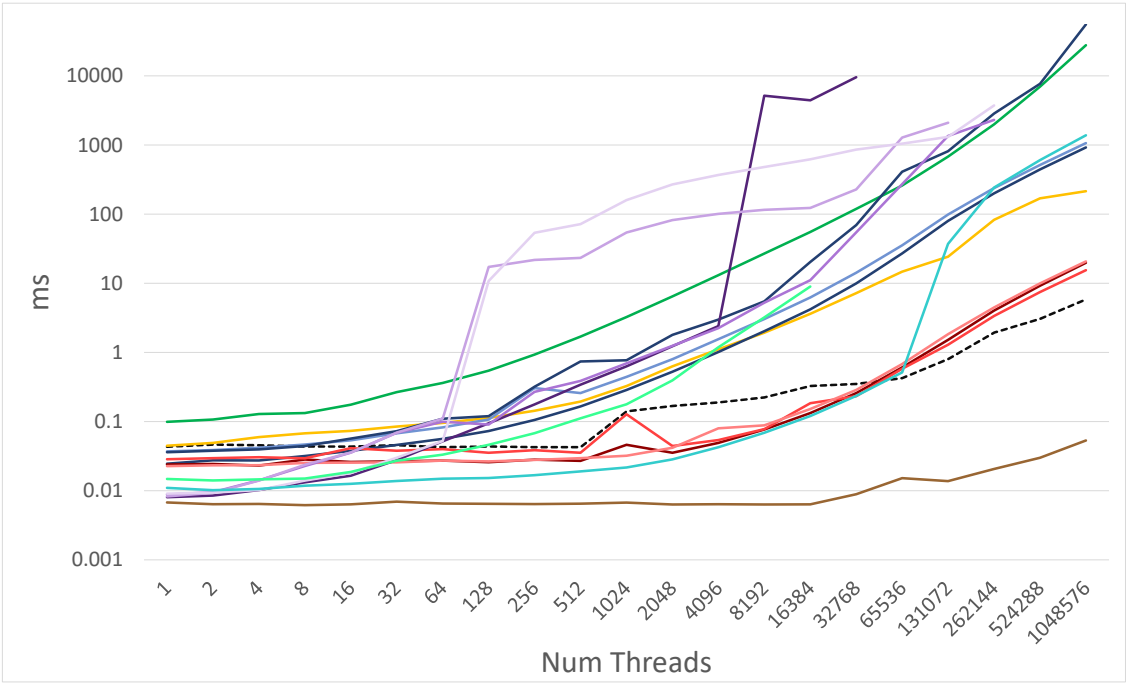


Figure 5.18: Work generation (4 B–4096 B per thread)

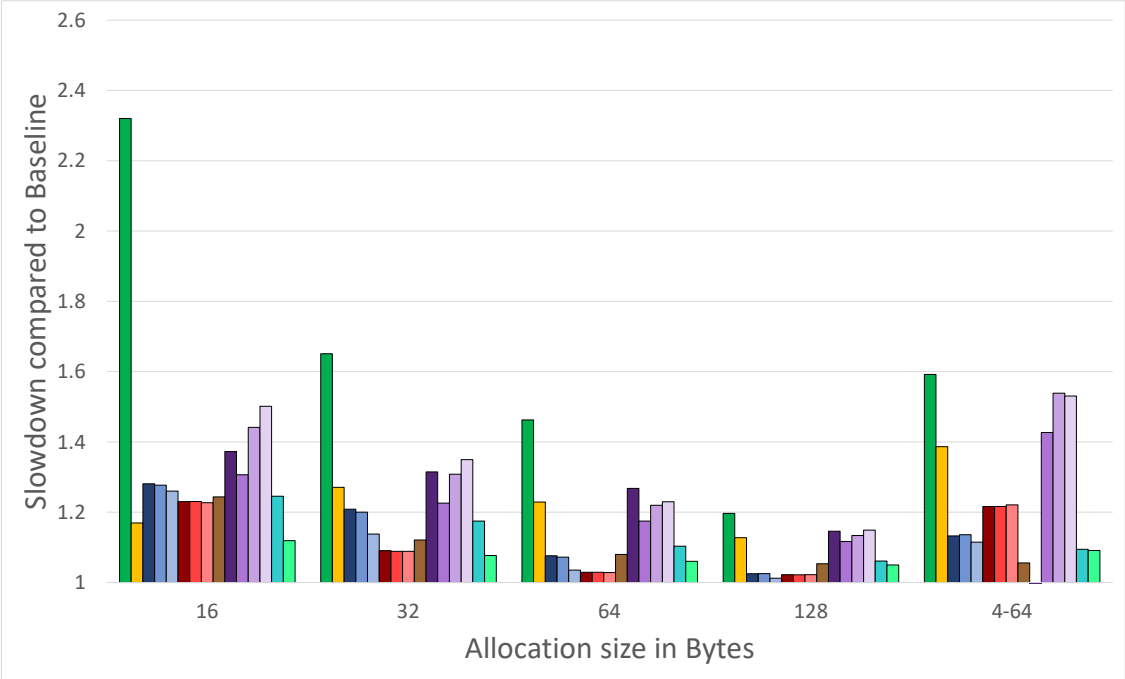


Figure 5.19: Write performance 100 000

Figure 5.20: Write performance to allocated memory compared to the *Baseline* (*Baseline* uses *CUB prefix-sum*) for the range 16 B–128 B (**5.19** for 100 000 allocations.

5.6.4.2 Memory Access Performance

On the GPU, not only allocation speed but also memory access speed is crucial. To evaluate alignment, we test the uniform and mixed case with 2^{17} allocations between 16 B–128 B. Each thread reads and writes to its assigned memory. As shown in Figure 5.19, *Ouroboros* stays closest to the fully coalesced baseline, closely followed by *XMalloc*, *ScatterAlloc* and *Halloc*. *RegEff* and the *CUDA-Allocator* show poor access times.

5.7 Discussion

Ouroboros introduces a novel approach for memory reuse based on *array-based queues*, improving upon the strengths of previous approaches. By expanding the data structures to allow for bulk allocation and virtualizing its base structure, we achieve efficient memory reuse and high allocation performance. By only keeping the current allocation state in memory, *Ouroboros*'s advanced *queueing* structures significantly trim down the memory overhead that comes with queue-based memory management.

We propose six configurations of *Ouroboros*, each managing pages, allocated from larger chunks of memory. The base queue operates either on pages directly or on chunks

holding pages, trading allocation speed for memory overhead. They can be realized fully in memory, virtualizing the queue by storing queues on chunks of memory using a small pointer array or just keeping pointers to the beginning and end of the queue. Each virtualization step reduces the inherent memory overhead at the cost of a slight decline in performance.

As we performed a thorough evaluation of all other memory managers as well, we now provide a short discussion on the merits of each tested approach.

The *CUDA-Allocator* offers a reliable option with a small register footprint. It works for any size and has very consistent performance, showing virtually no difference between *mean* and *median* performance. Unfortunately, its performance is comparatively weak overall, being consistently outperformed by all approaches for smaller allocations (up to around 2048 B, where its split is occurring) and only allocations larger than that favor it against a few other approaches. Furthermore, performance continuously increases with the amount of allocations and also appears to be dependent on the size of the manageable memory. Increasing this memory area is possible only by destroying the current context.

XMalloc is held back by its age, as it is not stable and fails most test cases, especially for larger allocation counts and mixed allocation sizes. It also represents an outlier in register footprint, which decreases its suitability even further.

ScatterAlloc is a very efficient dynamic memory manager with a clear focus on small allocations (performs clearly best for allocations ≤ 512 B and is competitive up to 2048 B). This also makes it the clear choice for any operation largely focused on smaller allocations, like the smaller synthetic workload case shown in Section 5.6.4.1. Larger allocations lag behind a bit and memory fragmentation also is not great due to scattering of memory accesses. Furthermore, increased thread contention affects *ScatterAlloc* more than others. *ScatterAlloc* is also very stable and can increase its manageable memory size at runtime. It also performs equally well for thread-based and warp-based allocations.

Halloc performs well until the point where it hands off to the *CUDA-Allocator*, staying reasonably close to *page-based Ouroboros* and *ScatterAlloc* for smaller allocations. It is clearly optimized towards warp-based allocations (outperforming *page-based Ouroboros* in this case), as thread-based performance is third best in the testset, but clearly behind the first two. It also splits its memory into two sections to accommodate larger allocations with the *CUDA-Allocator* and sacrifices some memory for increased performance, but performs second best when it comes to pure fragmentation.

RegEff comes in four different variants and shines when it comes to resource requirements, requiring the least amount of registers of all approaches. Unfortunately, performance is a mixed bag, with a large discrepancy between thread-based and warp-based performance (clearly favoring warp-based) and also very inconsistent performance, leading to significant differences between *mean* and *median* performance. Similar to the *CUDA-Allocator*, performance drops for increased saturation of the memory pool and fragmentation also is not great. Furthermore, not all variants are entirely stable and also none of them do return 16 B aligned memory, leading to issues with vector operations.

Ouroboros offers six variants of its allocator, which all excel when evaluating memory usage and fragmentation, but differ when it comes to performance. Its *chunk-based* variants outperform the *CUDA-Allocator* for allocations $\leq 2048\,\mathrm{B}$, but fall behind for larger allocations due to their two-stage access design. *Page-based Ouroboros* shows best performance overall, especially when considering thread-based allocations. Overall, *Ouroboros* favors thread-based allocations. It also shows some difference between *mean* and *median* performance, as re-use is drastically faster than allocating from an empty queue initially. Multiple instances can be stacked to allow for larger allocation sizes.

Our evaluation leads to the following conclusions:

- Thread-based Allocation
  - If an application mainly requires small allocations ($\leq 512\,\mathrm{B}$), *ScatterAlloc* is the clear choice with *Halloc* and the *page-based Ouroboros* staying close.
  - Larger allocations ($\leq 2048\,\mathrm{B}$) favor *Ouroboros*, followed by *Halloc*, *CUDA-Allocator* and *ScatterAlloc*.
  - Overall, *page-based Ouroboros* performs best, followed by *ScatterAlloc*, *Halloc* and the *CUDA-Allocator*.
- Warp-Based
  - *ScatterAlloc* performs best up to $4096\,\mathrm{B}$
  - *Halloc* also improves its performance, outperforming page-based *Ouroboros* up to $1024\,\mathrm{B}$
  - *Page-based Ouroboros* still is the best overall performer over the full tested range
- If fragmentation and memory utilization is of utmost concern, *Ouroboros* is the clear choice with *Halloc* a distant second.
- If register footprint is most important, then choosing one of the variants of *RegEff* might be sensible, but only if *warp-level programming* is used.
- If changes to the manageable memory size are required, then only *ScatterAlloc* and *Ouroboros* are suitable.
- Only the *CUDA-Allocator* and *Ouroboros* currently work on the newer GPU architectures with independent thread scheduling.
  - This may be crucial if support for warp-synchronous execution is dropped in future versions of CUDA.
  - This also currently limits any application using *XMalloc*, *ScatterAlloc*, *Halloc*, *RegEff* or *FDGMalloc*, as it would have to enforce warp-synchronous execution globally.

Considering the canonical example of work generation during a kernel, we showed that most approaches perform better than the canonical prefix-sum for smaller thread counts while *ScatterAlloc*, *Halloc* and *page-based Ouroboros* are a good choice even for large thread counts. We also showed that mature approaches like *ScatterAlloc* and *Halloc* still perform comparatively well, but that newer approaches, like *Ouroboros*, can leverage new hardware capabilities to both reduce fragmentation and increase performance, as it becomes less important to scatter memory accesses for increased performance.

Overall, considering our evaluation, performance worries with dynamic memory management on the GPU are exaggerated, as many approaches provide compelling performance with a straightforward usage model similar to CPU programming.

6

ouroGraph - Virtualized Queues for Dynamic Graph Management on GPUs

Contents

6.1 Introduction

faimGraph already considers many challenges posed by dynamic graph management and especially excels at updating the graph structure, be it inserting or deleting edges or even vertices. Furthermore, by offering two different internal storage formats in *AOS* and *SOA*, it also allows to perfectly adapt to the memory architecture of the GPU depending on the graph type at hand. Given the locality present on pages, even algorithmic performance overall is surprisingly good and the GPU-autonomy further strengthens the capabilities of the system.

Considering all these benefits, there still remain open challenges. For algorithms traversing the graph structure, the linked nature of the pages introduces additional complications. On the one hand, already existing algorithms are typically written for contiguous memory, hence would require manual adaptation to be able to run on the *faimGraph* data structure as well. On the other hand, even algorithms devised specifically for the linked structure have to make some concessions due to the memory layout, e.g., running a binary search on an adjacency requires some additional thought (and overhead) and once again, manual intervention. Furthermore, given the fixed page size within the system, a user always has to trade off memory access performance against memory efficiency. Small

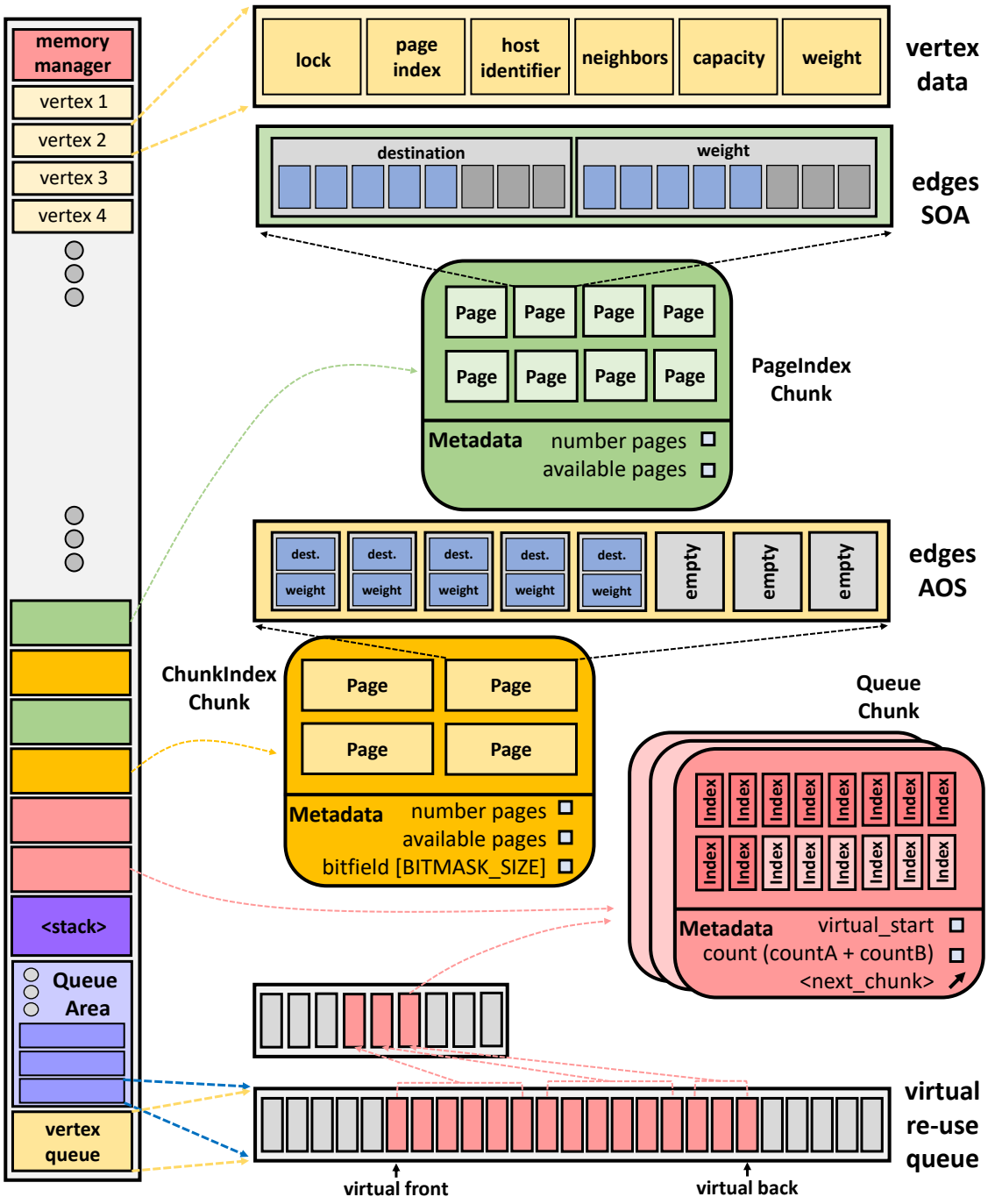


Figure 6.1: One large memory allocation holds a memory manager at the beginning as well as dynamic vertices growing from the bottom up. The end holds a vertex re-use queue, the base structure for potentially many queues as well as an optional area for temporary data while the rest is split into equally-sized chunks. Adjacency data is stored in either *AOS* or *SOA* format on power-of-two aligned pages.

pages keep close to the minimal requirements but incur heavy traversal for denser graphs. Large pages have the opposite effect, reducing traversal and increasing memory access performance at the cost of more memory being wasted. Choosing the right page size can be a challenge, especially for graphs with a larger distribution of adjacency lengths. This choice also heavily affects update performance for dynamic graphs, once again weighing higher update cost (and traversal) against memory efficiency as determined by the fixed page size.

To address these challenges, we present this final design, combining the benefits of contiguous memory per adjacency and GPU-autonomy, marrying the concepts found in *faimGraph* and *Ouroboros* into one, called *ouroGraph*. As *Ouroboros* already leaves the region right after its memory manager as linearly addressable memory, we can directly place our dynamic vertices into that region. Adjacency data is now stored on power-of-two aligned pages, allocated from larger chunks through the memory manager. This approach presents potential algorithms with one pointer to contiguous memory, identical to common storage formats like *CSR* or also both *cuSTINGER* and *Hornet*. After a description of the underlying design, we evaluate *ouroGraph* against *faimGraph* on the one hand, comparing update performance as well as algorithmic performance. Lastly, we also test graph initialization and edge updates against all other memory managers.

6.2 *ouroGraph*

A closer look at *Ouroboros*, see Figure 5.1, reveals certain similarities of the overall memory layout as found in *faimGraph*, compare Figure 4.1. At the beginning of the manageable memory, both designs place a memory manager unit and the *queues* are placed at the opposite end. *faimGraph* splits its memory from the top down into fixed-size pages, configurable at run-time, while all variants of *Ouroboros* split into large, fixed-size chunks, configurable at compile time and split into smaller, power-of-two aligned pages served to the user. The region after the memory manager, growing towards the pages/chunks, in both cases remains as linearly addressable memory.

This means that all dynamic graph functionality introduced with *faimGraph* natively translates to *ouroGraph* as well. A visualization of this design can be found in Figure 6.1. The start of the manageable memory region still houses a memory manager. Right after that, *ouroGraph* allows access to this region as linearly addressable memory. This fits the requirements of a fully-dynamic graph framework well, as dynamic vertices, growing from the bottom-up, can be placed there. At the opposite end, the queue area can be found, with multiple queues to serve all available page sizes within the system as well as a vertex queue for efficient re-use of previously deleted vertices. Right after the queue area, an optional, fixed-size stack region can be placed for temporary data required for updates or algorithms. The rest of the memory from the top-down is split at run-time into fixed-size chunks, which are then further split into power-of-two aligned pages usable as adjacency storage. This design is GPU-autonomous and fully dynamic, allowing for edge as well as

vertex updates, using the same update strategies (adapted for contiguous memory) and algorithms as presented in *faimGraph*.

The main difference to *faimGraph* comes from the specific adjacency storage format. While *faimGraph* uses fixed-size pages, linked together into a linked list, allowing for fast changes to the allocation state with good locality on each page, it still has some overhead in traversal, especially for denser graphs. *ouroGraph* deals with this issue by presenting potential graph algorithms with contiguous memory per adjacency. This significantly reduces the complexity of the update and graph algorithms and furthermore improves memory locality. It also allows arbitrary scheduling patterns and traversal strategies as one is not bound by arbitrary page sizes. Updating edges for a graph can be potentially faster or slower. This is dependent on the number of updates that lead to changes in the allocation state for *ouroGraph*, as this incurs higher cost through moving existing adjacency data instead of just adding or releasing a page to the memory manager, as in *faimGraph*. As long as the number of potential allocation state changes stays reasonably small, the improved access performance does improve performance overall.

As the number of allocation state changes grows, the overhead inherent with copying over adjacency data from a previous allocation this performance benefit shrinks and might even result in a slight overhead overall. We opted to not over-allocate the individual adjacencies (i.e., an adjacency requiring 32 B will get exactly 32 B) as we value memory efficiency over slightly increased update performance. Nonetheless, other approaches could be chosen. One potential candidate would allot overhead with exponential falloff, hence small adjacencies, which are at higher risk of reallocation, would get more potential space to reduce the occurrence of reallocations while larger adjacencies would get little to no overhead.

The same consideration applies to shrinking an adjacency once edges have been deleted. The chosen model once again opts for memory efficiency (i.e., an adjacency changing from 33 B to 32 B would be reallocated) but here other considerations would also be viable (i.e., only shrinking after a certain threshold has been passed). Reallocating an adjacency is done using vectorized loads/stores, but only affect performance for significant changes to a large number of adjacencies. Overall, this overhead is amortized by increased algorithmic performance as well as easier porting of existing algorithms and the increased memory efficiency, allowing the storage of larger graphs in the same initial allocation.

6.2.1 Vertex Updates

Vertices are stored exactly the same as for *faimGraph*, as detailed in Section 4.2.3.1. Each vertex has a number of properties (some of the optional depending on the graph type), as can be seen in Figure 4.4. To allow for vertex insertion without reallocation, the internal storage format is still *AOS* and new vertices are allocated by the memory manager either by re-using a free vertex index or by simply increasing the dynamic array of vertices, both achievable in *O(1)*. Updating vertices also follows the same procedures as detailed

in Algorithm 3 for insertion and Algorithm 4 for deletion. Compared to *faimGraph*, the individual steps become slightly more efficient, especially considering compacting the graph and finding vertex references, as no traversal is needed for each adjacency. Furthermore, the vertex re-use queue can also be virtualized, reducing the memory overhead to a fraction of the original footprint.

6.2.2 Edge Updates

While *ouroGraph* still supports both *AOS* and *SOA* layouts for the internal storage of adjacencies, that is the point where the similarities with *faimGraph* end. Instead of one fixed page size, linked together using the last 4 B, *ouroGraph* utilizes power-of-two aligned pages to store the adjacency data, irrespective of the internal storage format (as can be seen in Figure 6.1). As discussed already, we chose to assign the exact amount required for each adjacency without any over-allocation (except the natural overhead introduced by the alignment to power-of-two pages). This is a conscious choice based on our focus on memory efficiency.

We still offer all three update strategies as discussed in Section 4.2.5.1, so both *update-centric* and *vertex-centric* modes are possible. There are two differences to the general algorithm:

1. Since we can determine the final size of the adjacency after duplicate checking, we can, if necessary, first reallocate and transfer over the current state, and only then insert the updates. Compare that to *faimGraph*, where allocations happen on the fly as new updates are added and page boundaries are exceeded.

2. In case of *sorted vertex-centric* updates, instead of performing duplicate checking on-the-fly as in *faimGraph*, we also do this as a pre-processing step so that we know the final size of the adjacency and only need one reallocation request.

Edge deletion remains the same, either compacting from the back or respecting sort order, depending on the mode. If reallocation is required, then the update adjacency is built right on the new page with correct compaction. But once again, no traversal is required and at most one new page is allocated and one freed per adjacency.

6.2.3 Algorithms

Algorithms running on top of the graph structure stand to gain the most by the change in storage format. This entails a much smoother integration of existing work, as no traversal mechanics have to be implemented. Related work, especially work primarily devised for static graphs, typically assumes contiguous memory for its adjacencies. Removing the requirement for traversal semantics can hence lead to an easier porting of existing work to run on *ouroGraph*.

Second, without traversal, work balancing measures can much more effectively utilize the processing cores at hand and distribute them across the edges in question. *faimGraph* already provides an effort in the direction of work balancing, but this is limited to balancing over fixed-size pages used for adjacency storage. Considering graphs with a large distribution of adjacency lengths, this can be too broad of an approach to effectively reduce the imbalances in work. Furthermore, even when work balancing over pages works well, long adjacencies still require traversal first to reach the last page within an adjacency. Especially in denser graphs, although sufficient and balanced utilization might be provided, there still remains uneven, sequential traversal overhead as some threads take the first page in an adjacency while others need to reach the last page. This introduces additional divergence that can be detrimental to overall utilization and work balancing.

Third, basic building blocks like sorting or searching on an adjacency can be challenging to implement with traversal to resolve. *faimGraph* tries to counter this by already providing an update strategy retaining sort order, such that no additional sorting should be required. Furthermore, it slightly changes semantics during, e.g., binary search, where at first pages are traversed linearly, always checking just the first index to initially locate the page potentially holding an index and then the actual binary search is performed just on this page. Nonetheless, both cases show that additional effort and care has to be taken.

Last, depending on the choice for a fixed page size, the overall memory access pattern might be less than ideal. Especially when considering a size smaller than the 128 B cacheline size, as connected pages can be anywhere in memory, memory throughput can potentially suffer.

ouroGraph solves all of these problems by providing contiguous memory to the adjacencies, which alleviates traversal, makes porting trivial, usage of building blocks straightforward and also potentially results in less memory being transfered up and down from global memory through the cache hierarchy.

6.3 Evaluation

All performance measurements were conducted on an NVIDIA TITAN V (12 GB V-RAM) and an Intel Core i7-7700 with 32 GB of RAM.

Additional results on an NVIDIA RTX 2080Ti (11 GB V-RAM) can be found on GitHub. The framework is `CMake`-based and runs both on Linux and Windows. All given results were captured on Linux with `gcc` 10.2.0 using NVIDIA CUDA 10.2. Not all tested frameworks also work correctly with independent thread scheduling behavior introduced with the Volta generation of NVIDIA cards [41]. For these, we pass `compute_60` to the compiler to enforce warp-synchronous execution. Considering *ouroGraph*, $S|VA|VL$ denote standard, virtualized array-hierarchy and virtualized linked-chunk methods, $P|C$ define if page or chunk indices are stored. This results in the variants *Ouro-S-P*, *Ouro-S-C*, *Ouro-VA-P*, *Ouro-VA-C*, *Ouro-VL-P* and *Ouro-VL-C*. In cases which do not show if pages or chunk were used, the results were the same for both techniques.

Name	vertices	edges	adj. mean	adj. std. dev.	adj. max
luxembourg_osm	114599	239332	2.08	0.41	6
coAuthorsCiteseer	227320	1628268	7.16	10.63	1372
coAuthorsDBLP	229067	1955352	6.54	9.82	336
ldoor	952203	46522475	48.86	11.95	77
audikw_1	943695	77651847	82.28	42.45	345
delaunay_n20	1048576	6291372	6.0	1.34	23
rgg_n_2_20_s0	1048576	13783240	13.14	3.63	36
hugetric-00000	5824554	17467046	3.0	0.03	3
delaunay_n23	8388608	50331568	6.0	1.34	28
germany_osm	11548845	24738362	2.14	0.53	13
nlpkkt120	3542400	96845792	27.34	3.09	28
nlpkkt200	16240000	448225632	27.6	2.42	28
nlpkkt240	27993600	774472352	27.66	2.22	28
europe_osm	50912018	108109320	2.12	0.48	13

Table 6.1: Graph data set used for dynamic graph performance evaluation, taken from the 10th DIMACS Graph Implementation Challenge [5].

All frameworks were setup with 8 GB of manageable memory. Variants of *RegEff* were built with *warp-coalescing* turned off, as this did not work for any of the testcases. We use as a baseline a simple memory manager built on `atomics` on a shared `offset` (referred to as *Atomic*), but this is no true memory manager due to the lack of deallocation. We use a consistent color scheme throughout all plots to save on space; this color map can be seen in Figure 6.10.

Specifically evaluating dynamic graph management capabilities, we split this into two parts: once comparing against *faimGraph* directly, as can be seen in Section 6.3.1, and once against all other memory managers in Section 6.3.2.

6.3.1 Evaluation against *faimGraph*

To evaluate the performance in a real-world scenario, we adapted *faimGraph* to use *Ouroboros* and also the *CUDA-Allocator* to handle dynamic adjacency data. We will call the version using *Ouroboros ouroGraph* and the version using the *CUDA-Allocator Cuda-aGraph*. Using these allocators, adjacency data is stored on contiguous memory pages, compared to the linked-list of pages used in *faimGraph*. This speeds up the adjacency access and manipulation significantly and further simplifies the framework interface. The rest of the *faimGraph* framework remains unchanged. With this setup, we can also compare to other graph framework, like *aimGraph* [60], *cuSTINGER* [22], *Hornet* [13] and *GPMA* [46]. *faimGraph* has its queue initialized to 2 000 000 elements and uses 64 B pages. The algorithms were only modified to account for different adjacency traversals between *ouroGraph* and *faimGraph*. The used graph data set is listed in Table 6.1 .

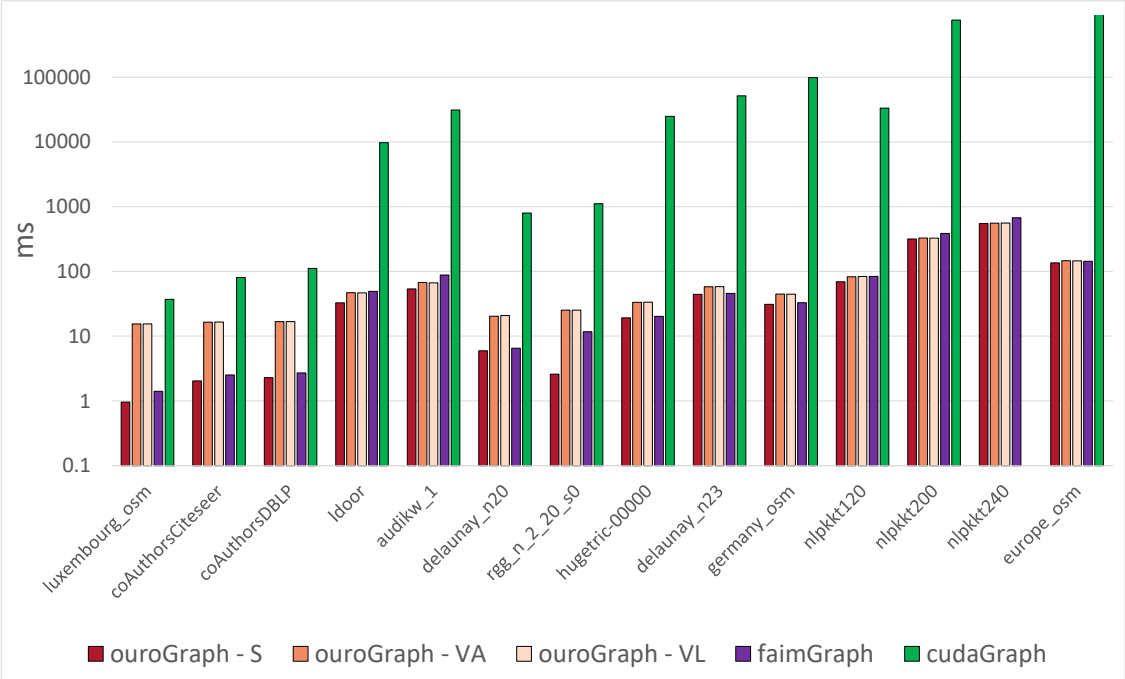


Figure 6.2: Graph Initialization Timing for all variants of *ouroGraph* as well as *faimGraph* and *CudaGraph*.

6.3.1.1 Initialization

faimGraph and *ouroGraph* are initialized similarly: both first determine memory requirements in parallel and then efficiently write adjacencies. For *CudaGraph*, the initialization cannot be sped up similarly as no knowledge about the data layout and underlying structures is available, which forces us to rely on a separate call to `malloc()` for each adjacency. The main difference in the initialization between virtualized *ouroGraph* and *faimGraph* is that *ouroGraph* flashes the complete memory with *deletion markers* first. This is necessary such that all chunks can immediately be used as *QueueChunks* without initialization, as this empty state is required of a queue. *faimGraph*, on the other hand, has to perform extra traversal and linkage of pages during the setup. These differences in the initialization are directly reflected in performance as shown in Figure 6.2.

Both *faimGraph* and *ouroGraph* outperform *CudaGraph* by an average of 1785×. For smaller and sparser graphs, *faimGraph* has the performance edge, as little to no page traversal is needed and it does not flash the complete memory as *ouroGraph* does. For denser and/or larger graphs, the difference shrinks or even reverses, as the included overhead of *ouroGraph* gets amortized by its more efficient memory access patterns. Concerning the memory footprint, it is clearly visible that both *virtualized* variants outperform standard *ouroGraph* and *faimGraph* in all cases, reducing the memory footprint on average by 23–32× compared to *faimGraph*. For smaller graphs, *faimGraph* has a small edge over

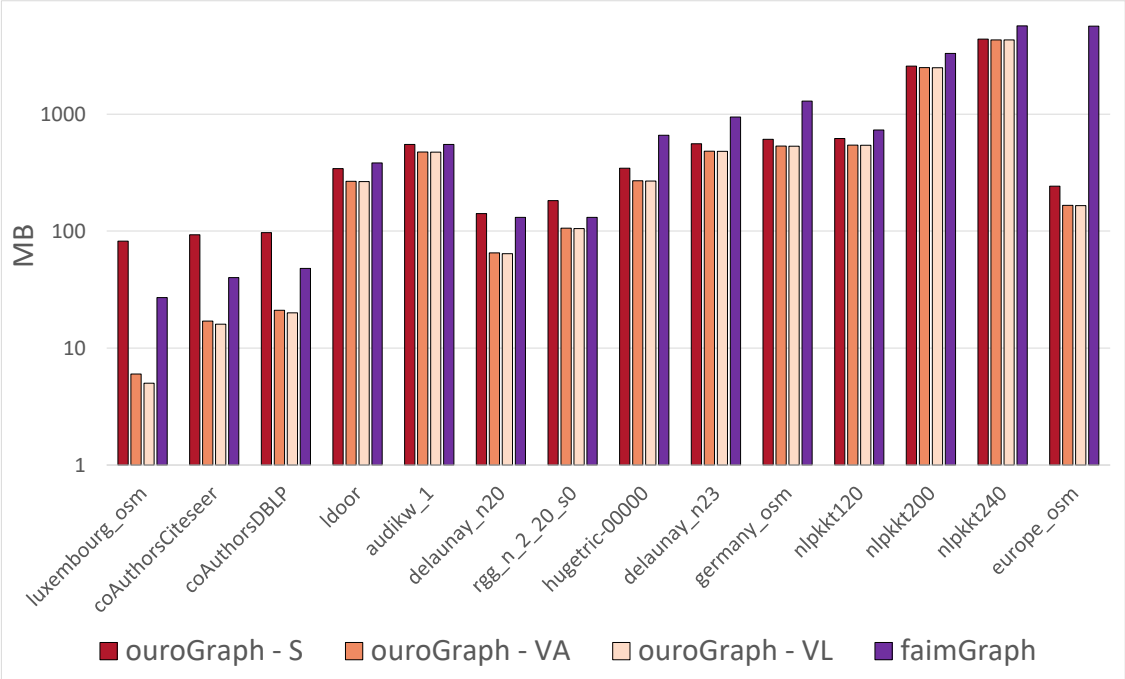


Figure 6.3: Graph Initialization Footprint for all variants of *ouroGraph* as well as *faimGraph* and *CudaGraph*.

the non-virtualized *ouroGraph*, but for larger (and especially sparser) graphs, standard *ouroGraph* also clearly outperforms *faimGraph*. The difference is largest for *europe*, a large graph with more than 50 million vertices, but a low adjacency degree. This increases the memory usage of *faimGraph* as the 64 B pages are too large for this graph, leaving a large portion of each page unused. *ouroGraph* packs each adjacency into nearly contiguous memory, as a fitting page size can be chosen for each.

6.3.1.2 Edge Updates

We investigate edge updates to gain insight into the real-world performance of the *ouroGraph* variants compared to *faimGraph* and *CudaGraph*.

To this end, we perform edge updates with a batch size of 100 000 and (1) randomized source as well as (2) with higher update pressure by fixing the source to a range of 1000 vertices. Note that *faimGraph* does not have to alter its initial adjacency data in the insertion case (pages are appended at the back) and remaining data in the deletion case (pages are just freed up at the back, remaining pages stay). *CudaGraph* and *ouroGraph* on the other hand copy complete adjacencies if the update results in a larger or smaller size compared to their current page size. This could be remedied by allowing some over-allocation on the existing allocations. Nevertheless, *ouroGraph* offers simplified and more efficient edge update procedures. The comparison in Figure 6.4–Figure 6.7 shows the clear

advantage in allocation performance of *ouroGraph* over *CudaGraph*.

A comparison to *faimGraph* necessitates different angles of interpretation, as updating a graph is multifaceted and allocation performance is only one important factor. For smaller and sparse graphs, we observe the *page-based* variants of *ouroGraph* outperform *faimGraph*. This is due to less frequent adjacency copies and/or moving smaller amounts of data for *ouroGraph*. Denser and larger graphs reverse this trend when the cost of copying individual adjacencies outweighs the more elaborate traversal of *faimGraph*. The results vary when comparing our approaches to *faimGraph*. For smaller and sparse graphs, adjacency copies happen less frequently and/or have to move smaller amounts of data. For these cases, we see the *page-based* variants of *ouroGraph* outperform *faimGraph*. For denser and larger graphs this trend reverses when the cost of adjacency copies outweighs the more elaborate traversal of *faimGraph*. *Chunk-based* variants of *ouroGraph* show less favorable performance compared to *faimGraph*, as (randomized) updates cause lots of chunks with just few pages to be enqueued. *CudaGraph* performs worst in all cases, being orders of magnitude slower for larger graphs.

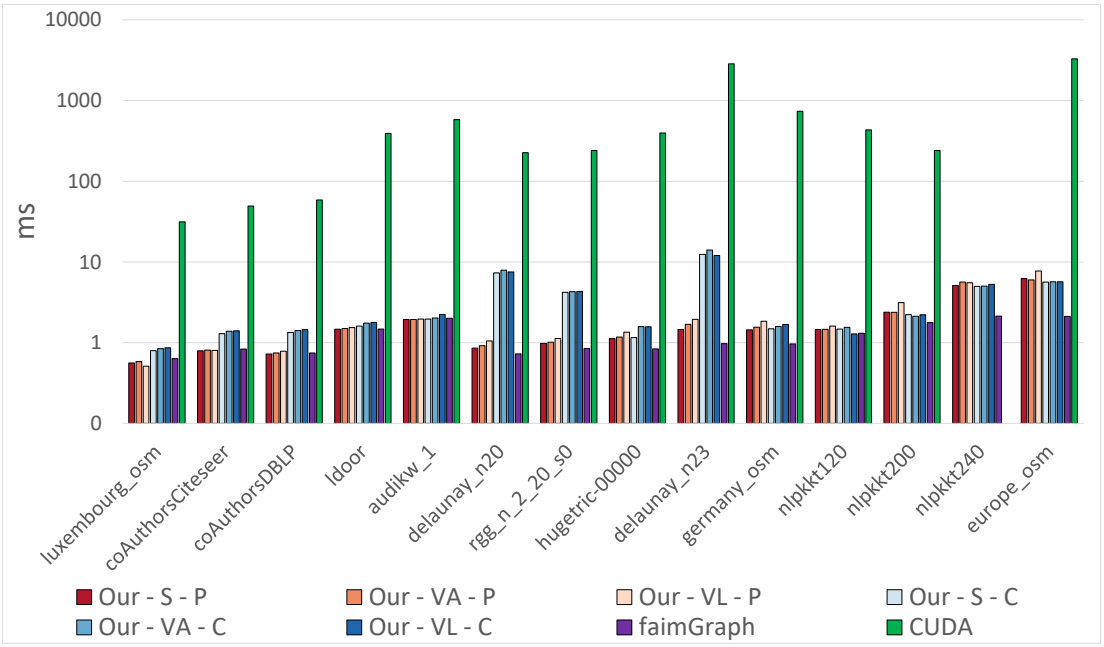


Figure 6.4: Edge insertion with 100 000 updates with random update source.

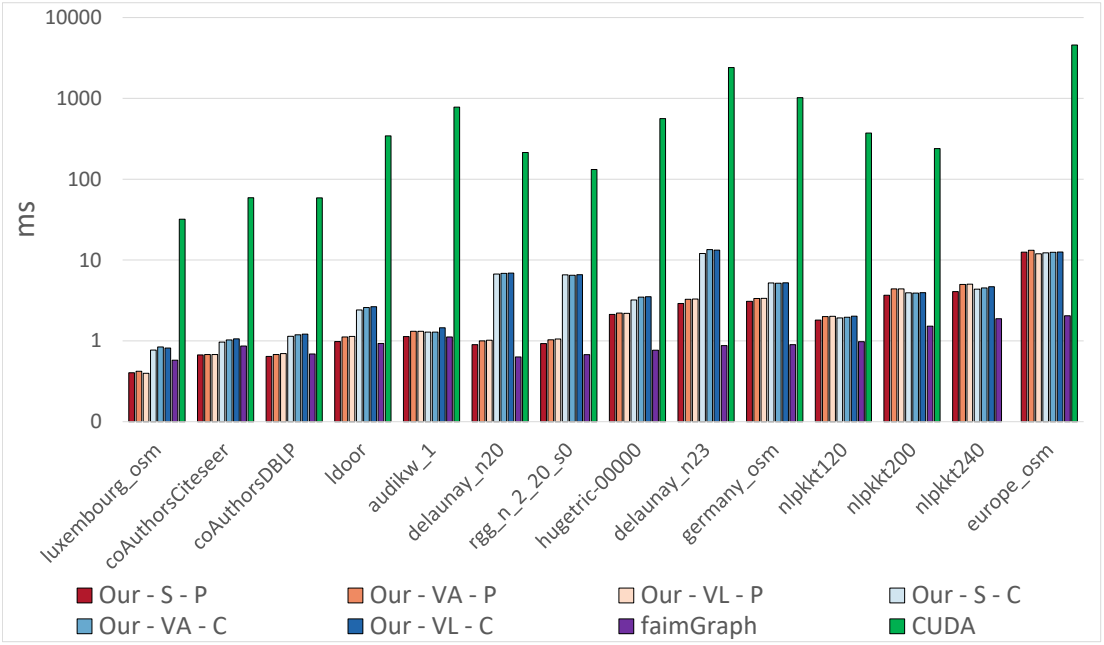


Figure 6.5: Edge deletion with 100 000 updates with random update source.

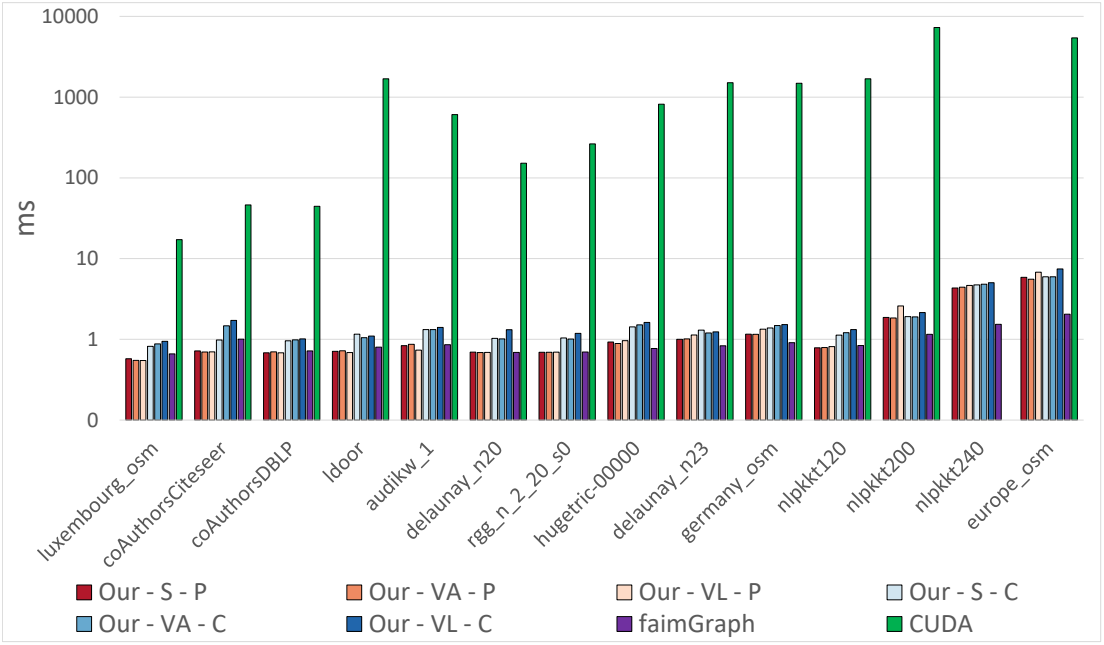


Figure 6.6: Edge insertion with 100 000 updates with source focused on a small range (1000) of vertices to simulate higher update pressure.

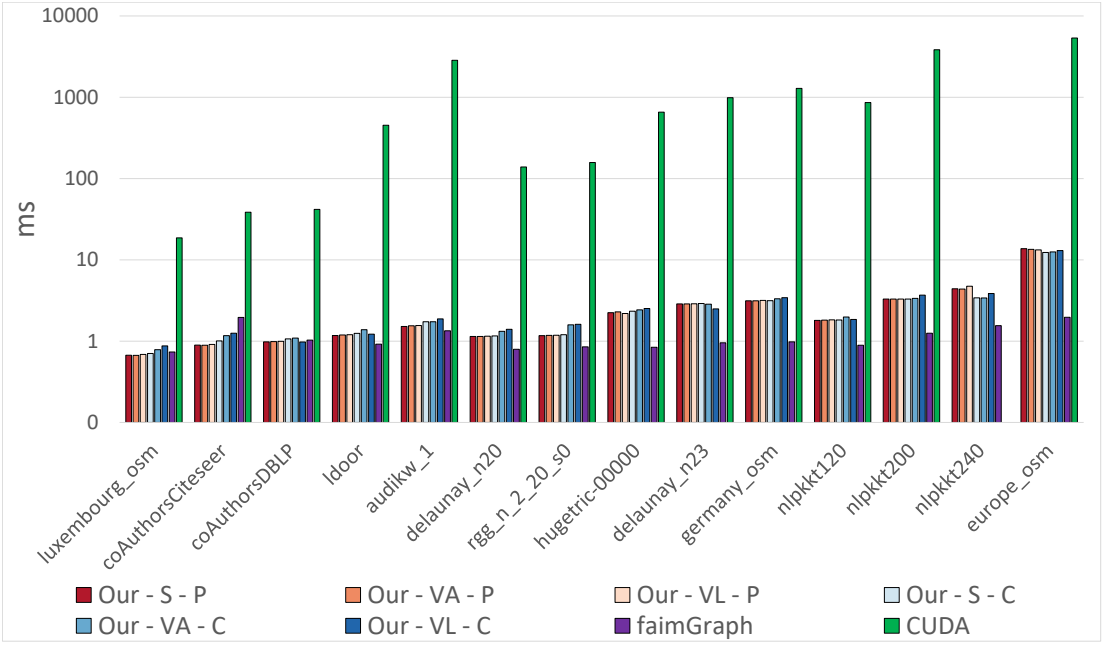


Figure 6.7: Edge deletion with 100 000 updates with source focused on a small range (1000) of vertices to simulate higher update pressure.

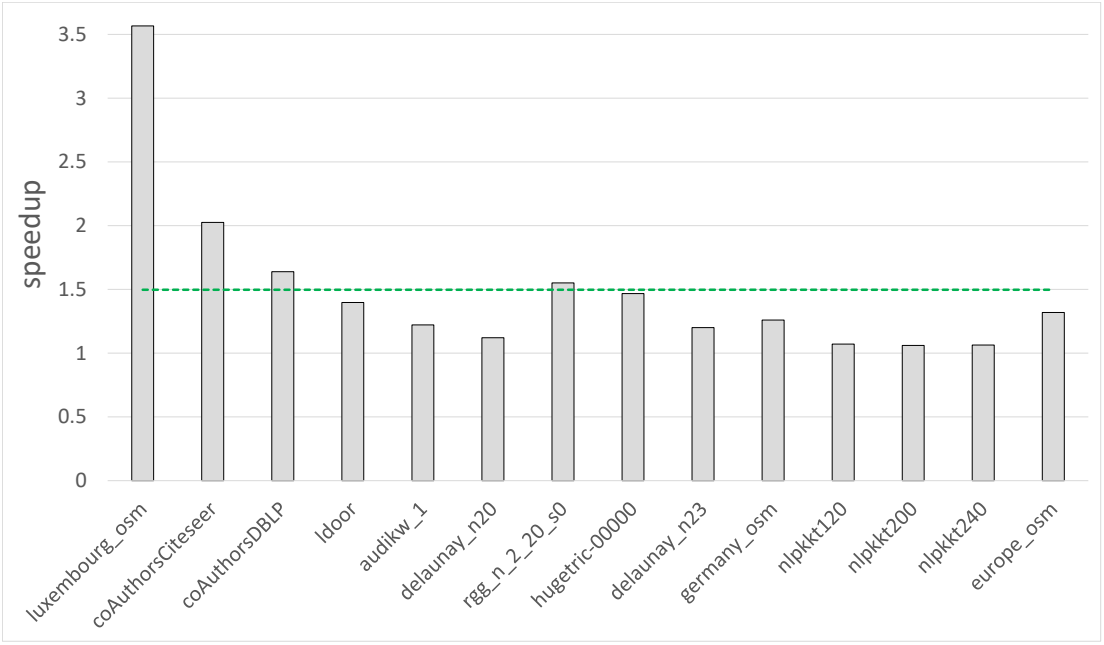


Figure 6.8: PageRank speedup of *ouroGraph* over *faimGraph*, highlighting the benefits of the contiguous memory layout of *ouroGraph*.

6.3.1.3 Algorithms

We also evaluate algorithmic performance of *ouroGraph* and *faimGraph* using *PageRank* and *Static Triangle Counting (STC)*. Presenting an algorithm with contiguous memory instead of partially-contiguous, linked memory (*faimGraph*) opens up performance potential, especially on the GPU. Furthermore, as *ouroGraph* eliminates the need for traversals, thread divergence and extra memory accesses are reduced. Lastly, as *ouroGraph* resembles more popular data structures like Compressed-Sparse-Rows (CSR), it is convenient to port efficient algorithm implementations. This makes it an ideal candidate to utilize efficient algorithm implementations of others without the need to convert the adjacency traversal to a proprietary structure.

PageRank Performance comparison using PageRank (Figure 6.8) clearly brings forward the benefits of *ouroGraph*—without thread divergence and page traversal, the GPU can get perfect memory access on adjacencies. Throughout the test set, performance benefits of *ouroGraph* are between 6–100% (50 % on average) with the contiguous adjacencies compared to the linked pages of *faimGraph*.

STC With *STC* we see the same trajectory. *ouroGraph* can significantly reduce the amount of excess memory accesses and improve code efficiency. Furthermore, we can bal-

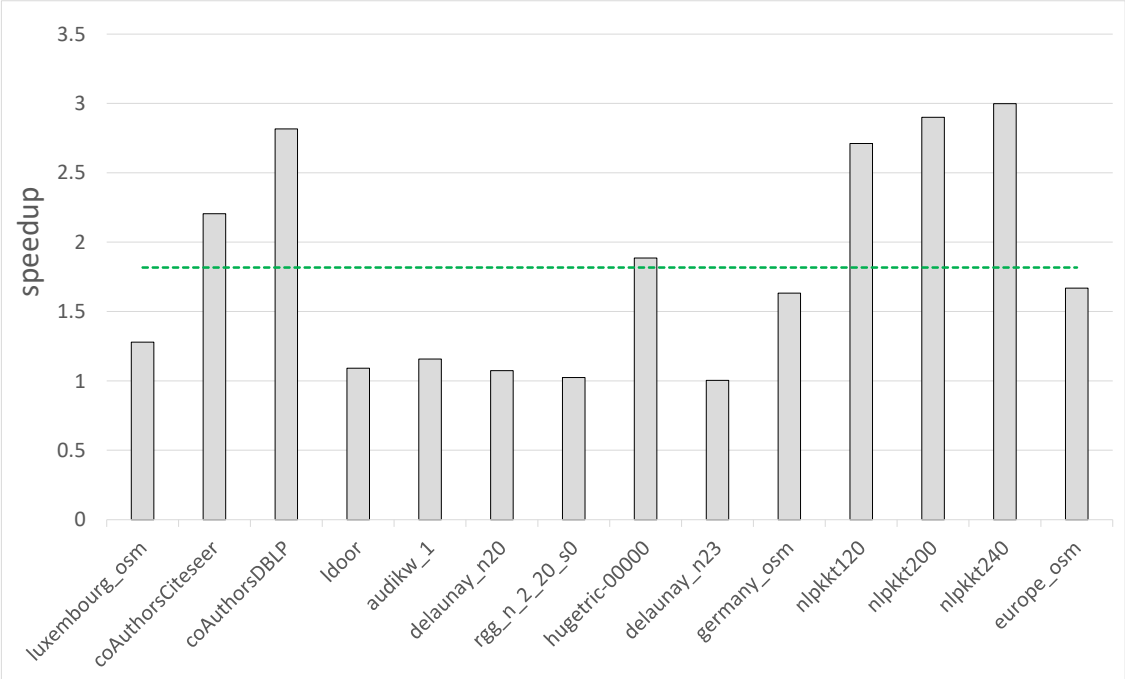


Figure 6.9: STC speedup of *ouroGraph* over *faimGraph*, displaying the benefits from contiguous memory, also allowing for more efficient binary search on the adjacency.

ance the number of workers per adjacency with greater freedom, as this is not constrained by the page size. Compared to *PageRank*, which allows for a comparatively simple setup, we can significantly reduce the register usage between *faimGraph* and *ouroGraph*, as the algorithm can work with indices alone and does not require larger iterators. The performance difference is again significant, see Figure 6.9.

On average, performance is 1.8 × higher (ranging from 1.03–2×) comparing *ouroGraph* to *faimGraph*. When starting whole warps per adjacency, we can more efficiently read in adjacency data and use *shuffle instructions* to communicate, regardless of adjacency size. This is limited in *faimGraph*—due to the fixed page size. Overall, *faimGraph* is a better option, only if update performance for large graphs is more important than algorithmic performance. *CudaGraph* falls short on update performance in all cases, leaving only good algorithmic performance as a positive. *ouroGraph* is close to *faimGraph*'s update performance, but significantly reduces memory requirements and boosts algorithmic performance. For smaller graphs or more elaborate algorithms, *ouroGraph* provides an enticing new approach to dynamic graphs on the GPU.

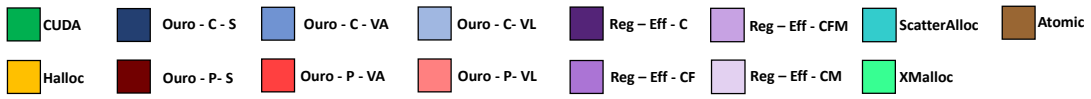


Figure 6.10: Color scheme used henceforth for all tested approaches.

6.3.2 Evaluation against other memory managers

Lastly, we also set up a simple graph framework capable of edge updates but not necessarily dynamic vertices using all memory managers. We test graph initialization performance for a smaller selection of graphs taken from the *DIMACS10 graph data set* [5] as well as updating the adjacencies.

6.3.2.1 Graph Initialization

Each adjacency is aligned to a power of two and the results of a setup from scratch. Allocating each adjacency individually, performance can be seen in Figure 6.11.

CUDA-Allocator performs worst in all scenarios, followed by the variants of *RegEff* and *chunk-based ouroGraph*. *Halloc* and *page-based ouroGraph* perform similarly, once again beaten by *ScatterAlloc*, as most graphs are sparse and require many small allocations. However, it is important to point out that since *ouroGraph*, given a certain graph to

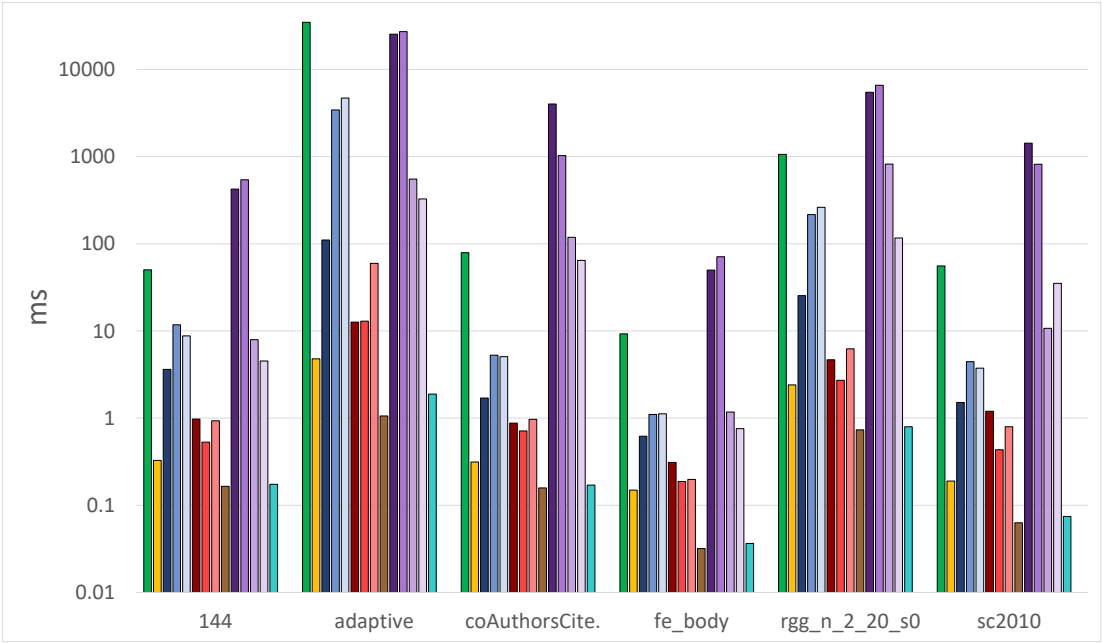


Figure 6.11: Dynamic graph initialization comparing the evaluated memory managers, each adjacency is allocated through the respective system.

initialize, results in a deterministic allocation state and can easily be setup in parallel with some precomputation of the requirements instead of this incremental build. This has been omitted in this case just to evaluate pure allocator performance during initialization. *ouroGraph* supports both a build from an existing graph (using precomputation) as well as an incremental build.

6.3.3 Edge Updates

We also consider updating the graph. As soon as an existing adjacency crosses over a power-of-two barrier during an allocation change, we allocate a new adjacency, move over the current data and free the old adjacency. We test multiple scenarios, including uniform updates as well as updates focused on a range of source vertices, to simulate more update pressure, which can be seen in Figure 6.12 and Figure 6.13. This testcase also highlights the ability to support concurrent allocations and deallocations, as each allocation change requires both allocation and deallocation. Once again, the *CUDA-Allocator* takes the most amount of time, followed by *RegEff* and then *chunk-based ouroGraph*. *Page-based ouroGraph*, *Halloc* and *ScatterAlloc* all perform equally well in this case.

One thing that sets *ouroGraph* apart as a memory manager for the scenario of dynamic graphs is the linearly addressable region at the beginning of the manageable memory area. This allows a seamless integration of the dynamic nature of vertices, as introduced in *faimGraph*, also in *ouroGraph*.

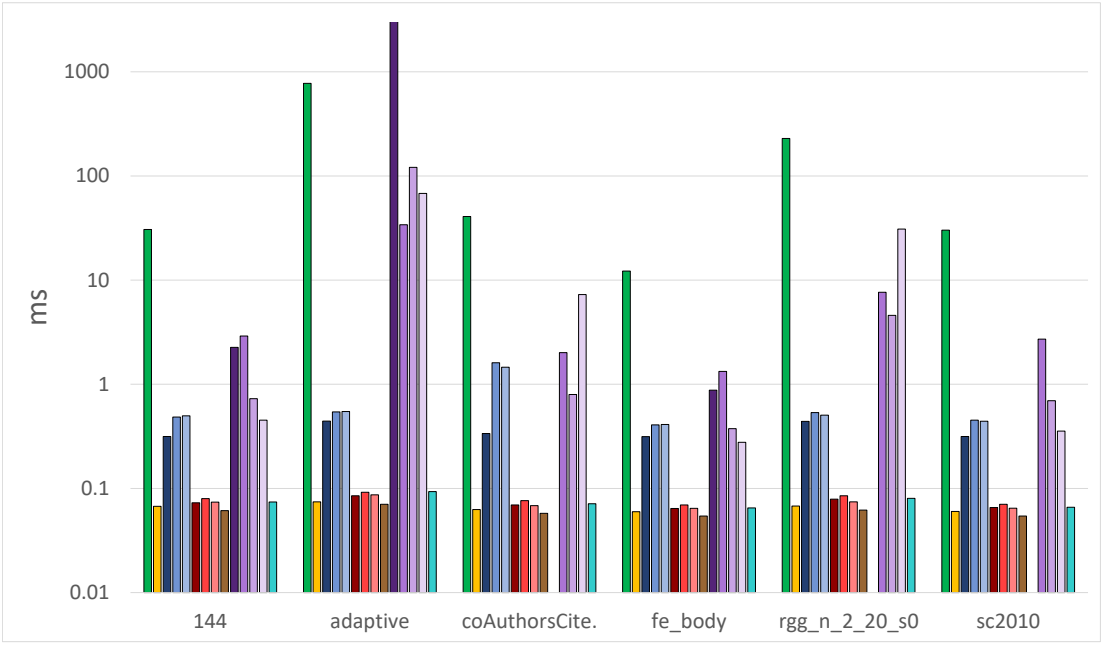


Figure 6.12: Edge insertion focused on a small range of source vertices with 100 000 updates

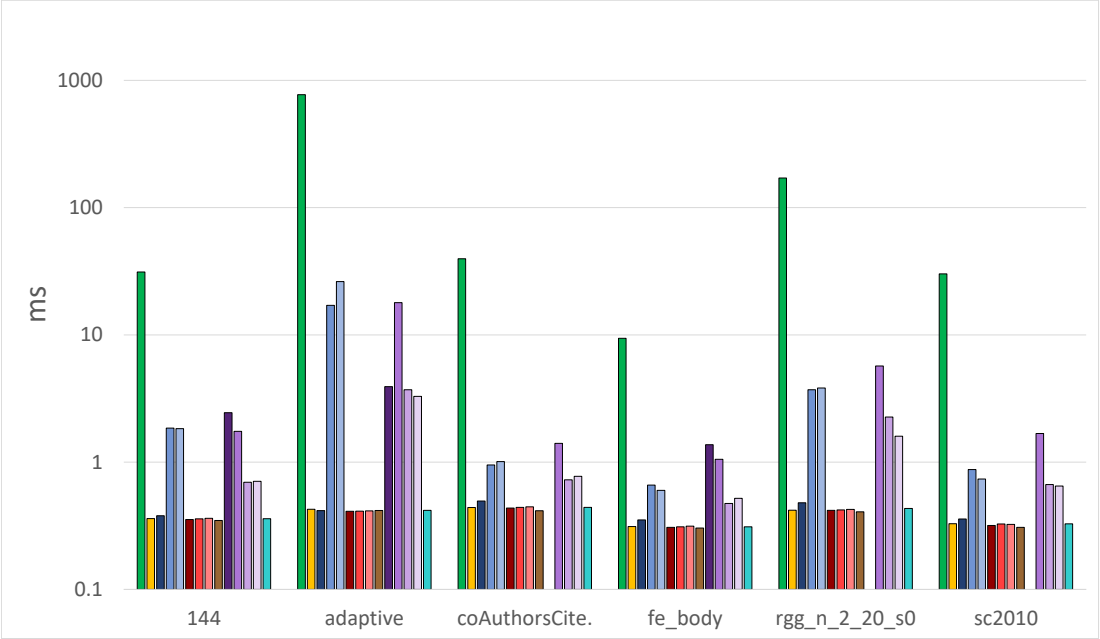


Figure 6.13: Edge deletion focused on a small range of source vertices with 100 000 updates

6.4 Discussion

ouroGraph combines the best of both worlds by incorporating *Ouroboros* into *faimGraph*, resulting in a fully-dynamic graph management system for the GPU with contiguous memory for its adajcencies. It offers a variety of techniques (*page-/chunk-based* management with two virtualization options) for its internal memory management, allowing the user to cater to a specific application domain. *Page-based* variants can be chosen for maximum performance during updates with outstanding memory efficiency, only bested by *chunk-based* variants at slightly reduced performance. Multiple instances of *Ouroboros* can be stacked within *ouroGraph* to allow for even larger adjacency sizes. In this case, a sensible solution could choose a performance-optimized, virtualized, *page-based* instantiation for smaller allocations, which occur more frequently and let a memory-optimized, virtualized, *chunk-based* variant deal with the few, large adjacencies found in some sparse graphs. Furthermore, work balancing for algorithms can be implemented on a much more fine-granular and adaptive level, as no page boundaries interrupt individual adjacencies.

Future work should continue at this exact point, as current work balancing measures still remain comparatively simple (though overall effective) and the possibilities are endless for more sophisticated work balancing measures. Secondly, currently *ouroGraph* always favors memory efficiency over update speed, which could potentially be solved by sensible or even adaptive thresholds for some additional storage per adjacency to reduce the effect of frequent reallocation of an adjacency close to a page boundary.

Overall, *ouroGraph* improves upon *faimGraph* primarily in the areas of memory efficiency (improving by 23 ×–32 × for the virtualized variants) as well as algorithmic performance (showing speed-ups in the range of 50 %–80 % on *PageRank* and *STC*), while staying competitive in the realm of edge updates even with a conservative bound on memory for each adjacency.

7 Conclusion

7.1 Conclusion

Dynamic graph and memory management on massively-parallel architectures like the GPU are topics that garnered interest in recent years. The ever increasing size of networks in highly dynamic application areas like communication, social-media and many more sparked the creation of a handful of dynamic graph management solutions. Recent hardware developments on modern *GPUs* (e.g., Independent Thread Scheduling and support for blocking algorithms) also rejuvenated interest in dynamic memory management on *GPUs*. As the GPU is becoming ever more popular as a general purpose processor due to limitations in hardware manufacturing as well as its suitability for problems with high data parallelism, we set out to provide viable solutions to both these domains.

Is it possible for a dynamic graph framework to run completely autonomously on the GPU? We answered this question with our first foray into the area of dynamic memory management culminating in *aimGraph*, a dynamic graph framework, running autonomously and independently on the GPU. It manages dynamic adjacency data in linked edge blocks, which can be allocated directly on the GPU from a simple memory manager. This design provides the best of both worlds for sparse graphs in terms of retaining memory locality within an edge block while still allowing efficient, dynamic changes to the adjacency size. Our tests showed that *aimGraph* is capable of millions of updates per second, outperforming *cuSTINGER*, especially in initialization performance.

Is it possible to design a dynamic graph framework for the GPU that truly allows for arbitrary growth and shrinking of both vertices and edges? The first evolution of our initial concept manifests itself in *faimGraph*, our fully-dynamic graph management solution on *GPUs*. It improves upon *aimGraph* in all aspects, introducing memory re-use queues for more efficient memory usage, different storage formats as well as intricate update strategies and work balancing for algorithms. Additionally, it was the first dynamic graph structure to also allow changes to vertices, enabling insertion and deletion of vertices using similar re-use structures as for the adjacency data. It outperforms *cuSTINGER*, *Hornet* and

GPMA in terms of initialization, update rates and algorithmic performance.

Is it possible to generalize dynamic memory management on the GPU, currently used only for specific user data, like vertices and edges, to general purpose memory management and its management structures? Inspired by the concepts of GPU-autonomy and memory re-use on the GPU, we broadened the scope and crafted a general purpose memory manager for *GPUs*, called *Ouroboros*. At its core, it uses once again a memory manager and queues on the GPU directly to manage the available memory. As memory efficiency is always our focus, we virtualized the basic re-use queues in two variants and store the actual queue content on dynamic chunks re-usable within the system, minimizing the memory overhead common with array-based queues. As part of an extensive survey of the current state of dynamic memory managers on the GPU, we showed that variants of *Ouroboros* clearly perform best regarding memory efficiency and fragmentation and *page-based* variants of *Ouroboros* even perform best over large allocation ranges.

Is it possible to use this generalized approach for dynamic graph management as well to increase memory locality as well as algorithmic performance? Lastly, we incorporated all dynamic graph functionality introduced in *faimGraph* into *Ouroboros*, called *ouroGraph*. It fuses the best of both approaches, storing adjacency data on power-of-two aligned pages and allows for dynamic changes to vertices as well. Based on this approach, memory efficiency and algorithmic performance get a sizeable boost and porting existing algorithms to *ouroGraph* also was simplified at comparable or slightly lower update performance.

Overall, we believe that our efforts in the areas of dynamic graph and memory management provide an initial big step towards feasible solutions for highly dynamic problems on the GPU. Our evaluation suggest that our solutions excel in memory efficiency and also show great performance when accessing memory, updating vertices or adjacencies or running algorithms. By making all of our efforts open source, we hope to inspire further work in the areas of dynamic graph and memory management and thereby increase the utility of the GPU for many more problem domains.

7.2 Future Work

There still remain many interesting areas to explore in the future. In terms of dynamic graph management specifically, most frameworks can already harness the parallel capabilities of one GPU, but automatically scaling to multiple *GPUs* still is a manual task. Especially when considering problems like locality over multiple *GPUs* and how to distribute both the graph and the workload remains a challenging proposition. In a dynamic environment, such a framework would also have to consider the challenges presented by dynamic changes, as some nodes in such a configuration could end up with ever increasing shares of the graph. As we already provide a vertex mapping system, transferring adjacencies between *GPUs* could already be integrated seamlessly, but many considerations, including where to place data to reduce access overhead and memory locality, have to be focused on.

Furthermore, all existing work considers algorithms and graph management tasks to be disjunct during execution, occurring only one after the other. Similarly for multiple algorithms running on the graph, these typically also execute one after the other and concurrent execution already can be a challenging issue for efficient task scheduling. This is especially true when considering that these algorithms come with very different properties. Some might finish quickly, i.e., *PageRank*, while others might be more involved and exhibit longer runtimes. Running multiple algorithms can certainly be more efficient to fully utilize the parallel capacities of a GPU, but in this case already, prioritization might be an essential component, as some algorithms provide crucial insight while others are not time critical. Additionally, especially when algorithms take a significant amount of time to complete, there might be little time to interject updates to the graph structure. Changes to the graph could also originate from an algorithm, adding an additional layer of complexity. Providing support for concurrent graph management and algorithm execution for truly dynamic scenarios represent a significant challenge to be solved in the future.

Regarding dynamic memory management, our page-based queue shows the best performance overall, but limits chunk reusability at this point in time. The problem stems from the fact that each page can be stored in any location within the queue and these positions are not trivial to find and invalidate in case of taking out a full chunk from the queue. Future efforts could investigate possibilities for queue compaction, which would need to find a way to invalidate indices in a queue without interrupting concurrent execution.

7.2.1 NVIDIA API & Hardware Features

Considering the recent API introduction of group-based mechanics, adding group-based allocation strategies would be a vital addition to our efforts. Our current efforts in *aimGraph*, *faimGraph* as well as *Ouroboros* put an emphasis on thread-based computation. This strategy has paid off as more hardware support from NVIDIA through *Independent Thread Scheduling* has greatly increased performance for such workloads. Scheduling branches instead of warps furthermore helps performance in case of traversing adjacencies of greatly varying degrees. Nonetheless, many applications still perform best executed by multiple, cooperating threads. This is true for denser graphs especially (all our experiments for graph-based processing typically switch to warp-based processing after a certain average adjacency threshold) but fine-grained worker size selection could also result in performance benefits for power-law graphs. In terms of memory management, it would reduce the pressure on the memory manager during allocation. Building on the *Cooperative Groups API*, this flexibility can be neatly integrated into all previously mentioned frameworks.

All graph frameworks currently do not leverage shared memory in any major extent. When processing updates with multiple threads, especially if sort order is maintained, doing these operations in shared memory cooperatively and only load/store to global memory

once could greatly improve performance as well. Recent API introductions even allow for staging the loading of global memory into shared memory, opening up the possibility to interleave memory transfers and computations even more efficiently.

Building on *Independent Thread Scheduling*, we already utilize the ability to gather currently active threads and let them cooperate at a certain section of the program. *Ouroboros* uses these features already for its synchronization primitive as well as for the queues, resulting in potentially more memory access locality in memory. Incorporating these techniques into *faimGraph* (and *aimGraph*) could also result in increased performance, especially when accessing the re-use queues.

Lastly, integrating *Ouroboros* into the vendor-provided page-table system could open up access to system memory through page faults. This could vastly increase the amount of memory available to the system and thereby allow for the storage and management of even larger graphs, at least partly, on the GPU. This could also lead to a possible solution in case vertices can be assumed static, where no large, initial allocation is required but all chunk allocations can be served by the page-table system, reducing the initial static requirements. Furthermore, the recent introduction of *virtual* memory could also be leveraged to decrease the initial memory footprint of the system. In this case, one could reserve a large range of addresses up front but only map memory blocks to these addresses on demand. This way, instead of always having to trade-off between the initial size and potential reinitializations, all of this could be handled at run-time.

A

List of Publications

My work at the Institute of Computer Graphics & Vision at the Graz University of Technology led to the following peer-reviewed publications.

2017

- **[Best Student Paper Award]** M. Winter, R. Zayer, and M. Steinberger. Autonomous, Independent Management of Dynamic Graphs on GPUs. In *2017 IEEE High Performance Extreme Computing Conference (HPEC'17)*, pages 1–7, Waltham, Massachusetts, USA, 2017. University of Technology, Graz, IEEE, `https://doi.org/10.1109/HPEC.2017.8091058`

2018

- M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger. FaimGraph: High Performance Management of Fully-Dynamic Graphs under Tight Memory Constraints on the GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, Dallas, Texas, USA, 2018. IEEE Press, `https://doi.org/10.1109/SC.2018.00063`

2019

- M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger. Adaptive Sparse Matrix-Matrix Multiplication on the GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 68–81, New York, NY, USA, 2019. Association for Computing Machinery, `https://doi.org/10.1145/3293883.3295701`

- D. Toedling, M. Winter, and M. Steinberger. Breadth-First Search on Dynamic Graphs using Dynamic Parallelism on the GPU. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2019, https://doi.org/10.1109/HPEC.2019.8916476

2020

- M. Winter, D. Mlakar, M. Parger, and M. Steinberger. Ouroboros: Virtualized Queues for Dynamic Memory Management on GPUs. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, New York, NY, USA, 2020. Association for Computing Machinery, https://doi.org/10.1145/3392717.3392742

- M. Parger, M. Winter, D. Mlakar, and M. Steinberger. SpECK: Accelerating GPU Sparse Matrix-Matrix Multiplication through Lightweight Analysis. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 362–375, New York, NY, USA, 2020. Association for Computing Machinery, https://doi.org/10.1145/3332466.3374521

- [**Best Paper Award**] D. Mlakar, M. Winter, P. Stadlbauer, H.-P. Seidel, M. Steinberger, and R. Zayer. Subdivision-Specialized Linear Algebra Kernels for Static and Dynamic Mesh Connectivity on the GPU. In *Computer Graphics forum Volume 39, Issue 2*, EG '20, page 0, 2020, https://doi.org/10.1111/cgf.13934

2021

- M. Winter, M. Parger, D. Mlakar, and M. Steinberger. Are Dynamic Memory Managers on GPUs Slow? A Survey and Benchmarks. In *Proceedings of the 26th Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pages 68–81, New York, NY, USA, 2021. Association for Computing Machinery, https://doi.org/10.1145/3437801.3441612

- D. Mlakar, M. Winter, M. Parger, and M. Steinberger. Speculative Parallel Reverse Cuthill-McKee Reordering on Multi- and Many-core Architectures. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IPDPS '21, page 0, 2021, https://doi.org/10.1109/IPDPS49936.2021.00080

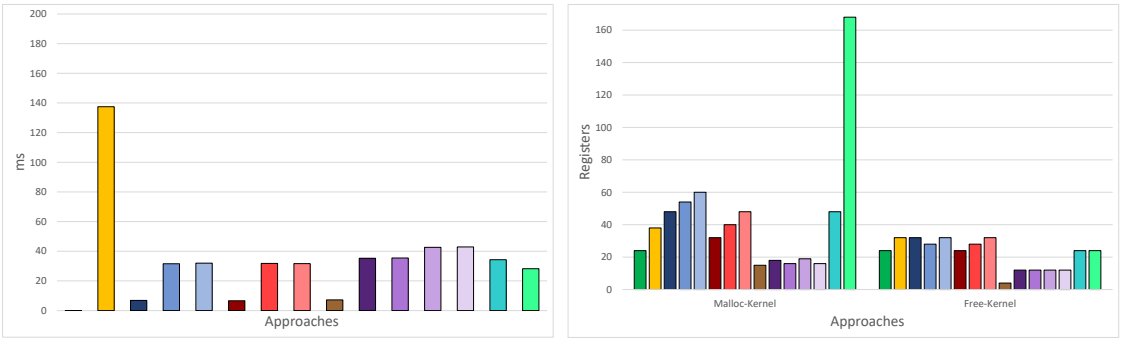
B Plots

B.1 Memory Manager Survey

This appendix holds all results captured during the evaluation of dynamic memory managers on the *GPU*, and those results can also be found in the GitHub repository. All performance measurements were conducted on an NVIDIA TITAN V (12 GB V-RAM) and an Intel Core i7-7700 with 32 GB of RAM and took around 600 h (roughly $3\frac{1}{2}$ weeks) to complete. We also tested on an NVIDIA RTX 2080Ti (11 GB V-RAM) and an Intel Core i7-7700 with 32 GB of RAM and took also around 600 h (roughly $3\frac{1}{2}$ weeks) to complete (these results can be found in the repository (GitHub)). Figures tagged with **[Titan V]** were run on the TITAN V and **[2080Ti]** denotes results gathered on the 2080Ti. All plots use a consistent color scheme, which can be seen in Figure 5.5.

B.1.1 Results on NVIDIA TITAN V

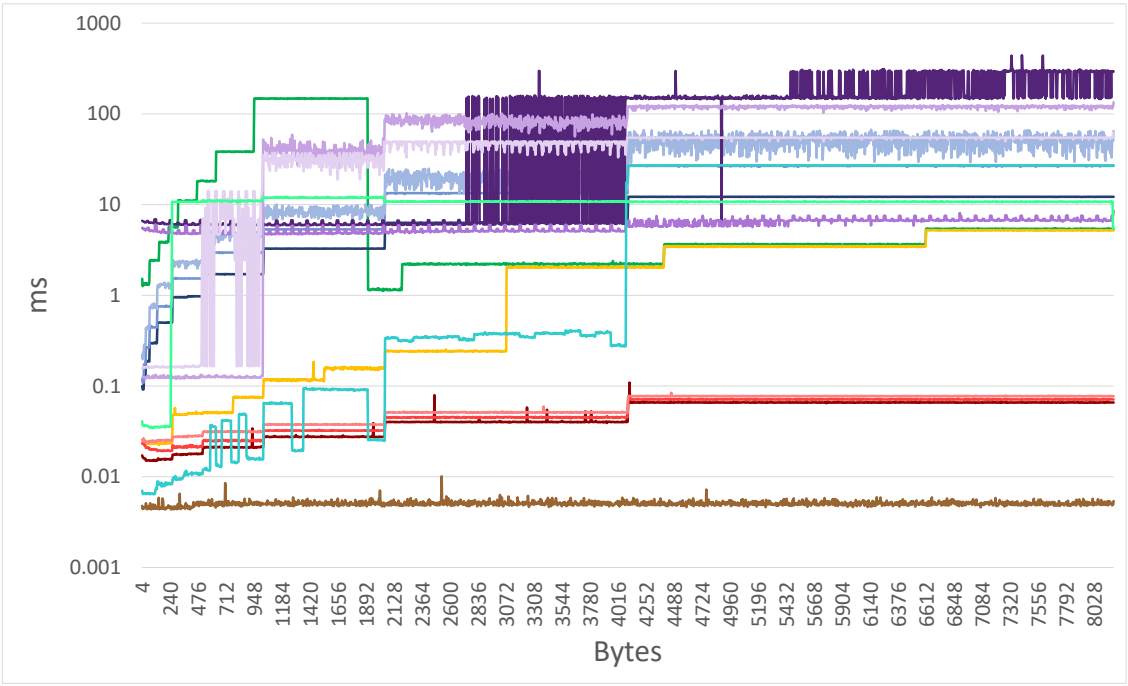
- Figure B.1 shows initialization performance as well as register requirements for *malloc* and *free*.
- Figure B.2 shows thread-based allocation and deallocation performance for 10.000 and 100.000 allocating threads, both *mean* and *median* performance is shown.
- Figure B.3 shows warp-based allocation and deallocation performance for 10.000 and 100.000 allocating warps, both *mean* and *median* performance is shown. Per warp, only one thread allocates or deallocates memory.
- Figure B.4 shows thread-based, mixed allocation and deallocation performance for 10.000 and 100.000 allocating threads, both *mean* and *median* performance is shown. Allocation stem from a range between 4 B to 4 B–8192 B.
- Figure B.5 shows allocation performance scaling for threads in the range 2^0 to 2^{20} and the byte range between 4 B–8192 B.



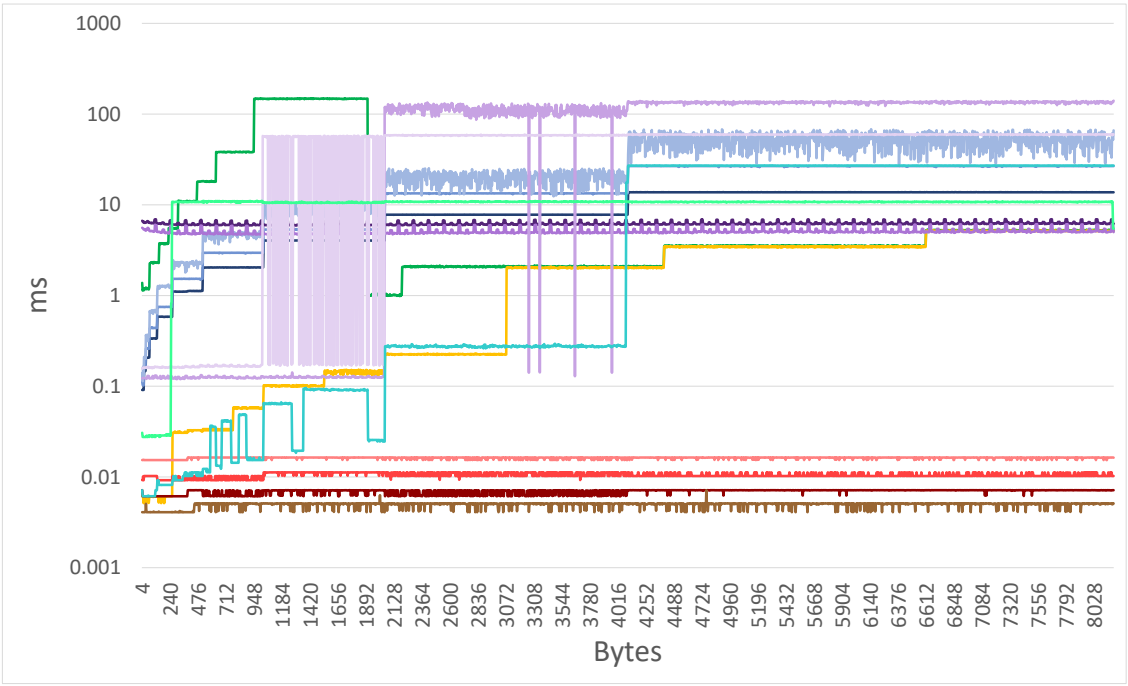
(a) Initialization Timing (b) Register Requirements

Figure B.1: Initialization performance as well as register requirements for *malloc* and *free*

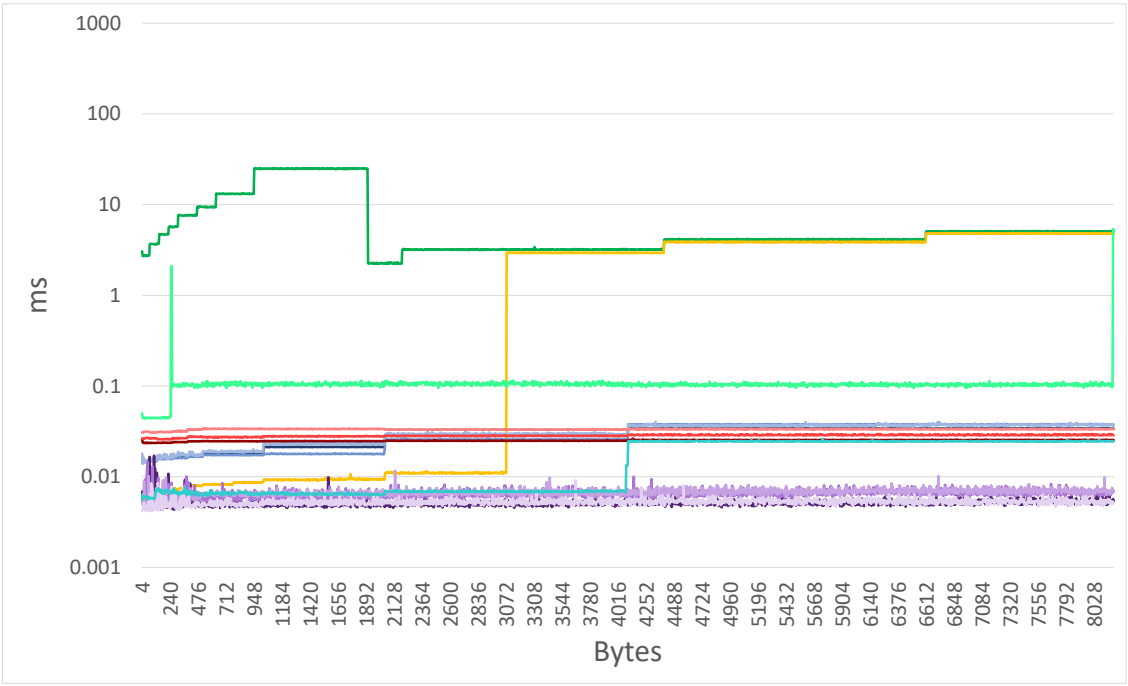
- Figure B.6 shows allocation performance scaling for threads in the range 2^0 to 2^{20} and the byte range between 4 B–8192 B.
- Figure B.7 shows various plots, including initial and static fragmentation, out-of-memory performance, two synthetic work generation testcases for the ranges 4 B–64 B and 4 B–4096 B as well as memory access performance over the range 16 B–128 B as well as for mixed allocations in the range 4 B–64 B.
- Figure B.8 shows dynamic graph initialization as well as edge updates to the graph. 100.000 updates are applied to the graph, either uniformly distributed on the graph or focused on a source range of 1.000 vertices.



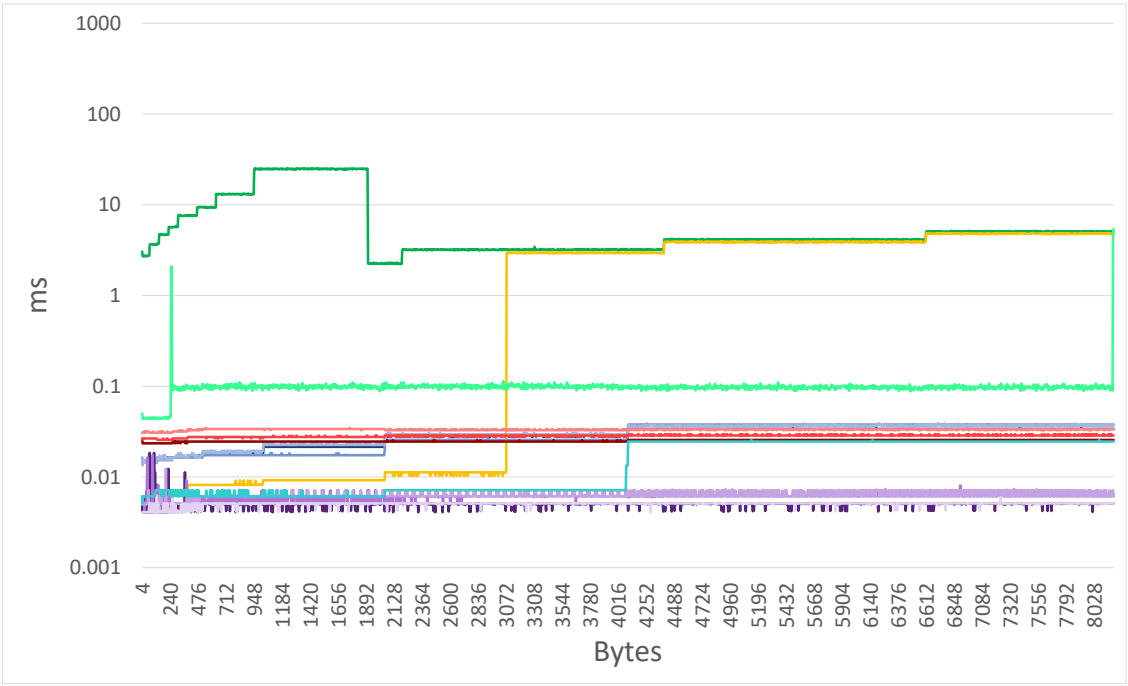
(a) Allocation performance 10K - **mean**



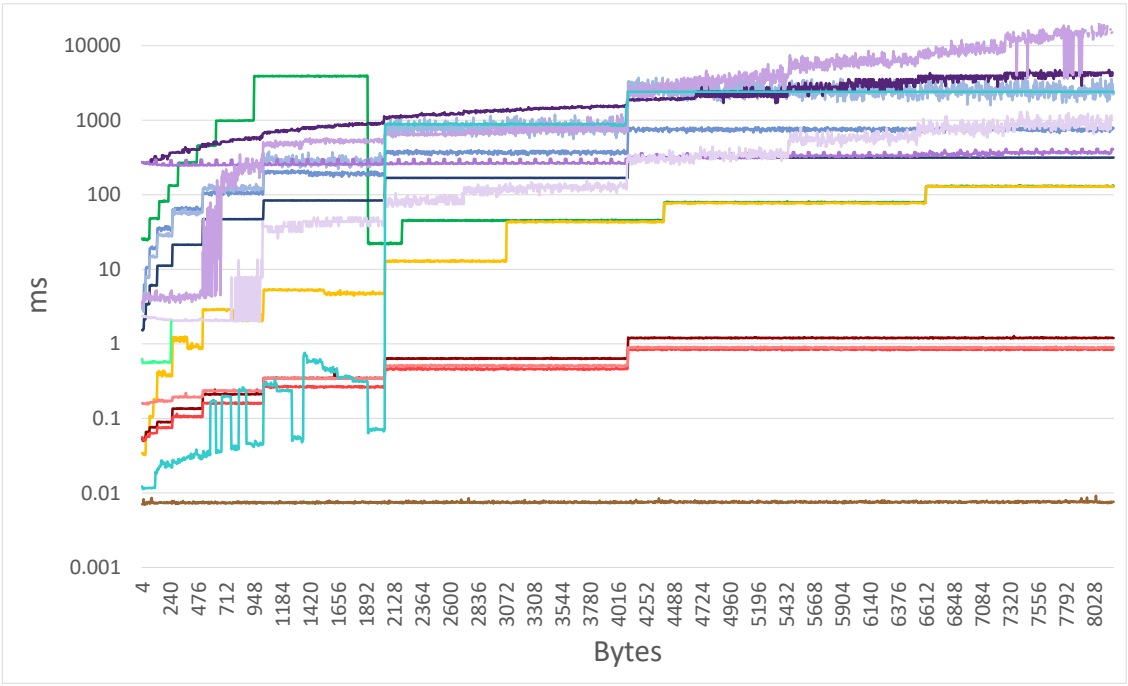
(b) Allocation performance 10K - **median**



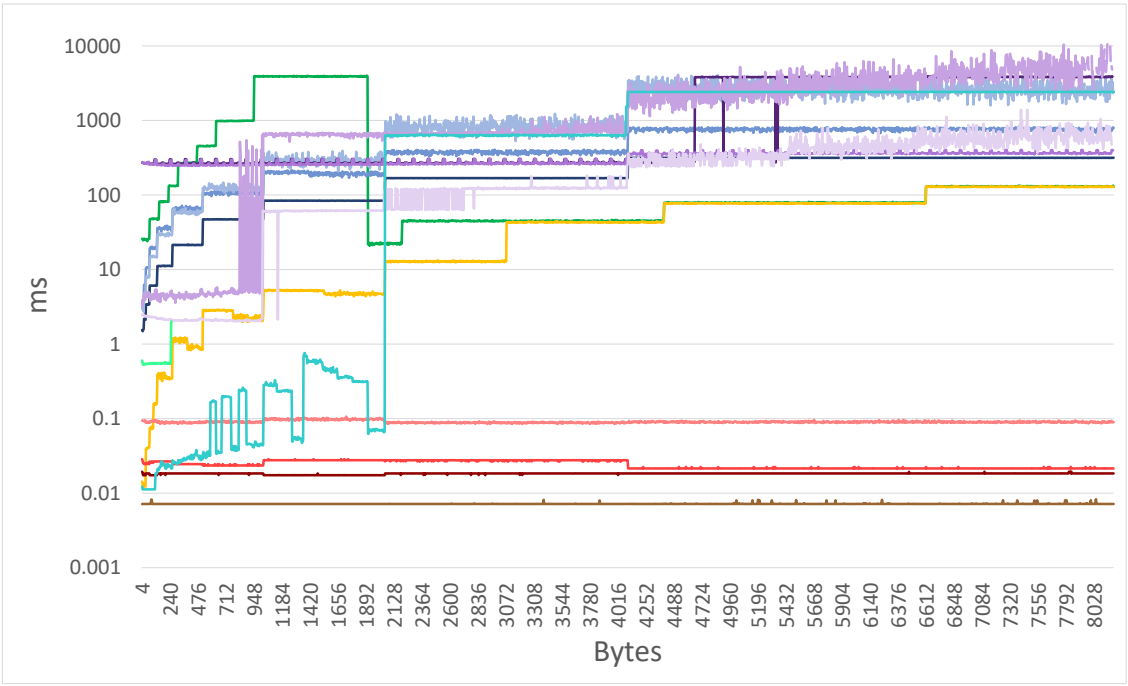
(c) Deallocation performance 10K - **mean**



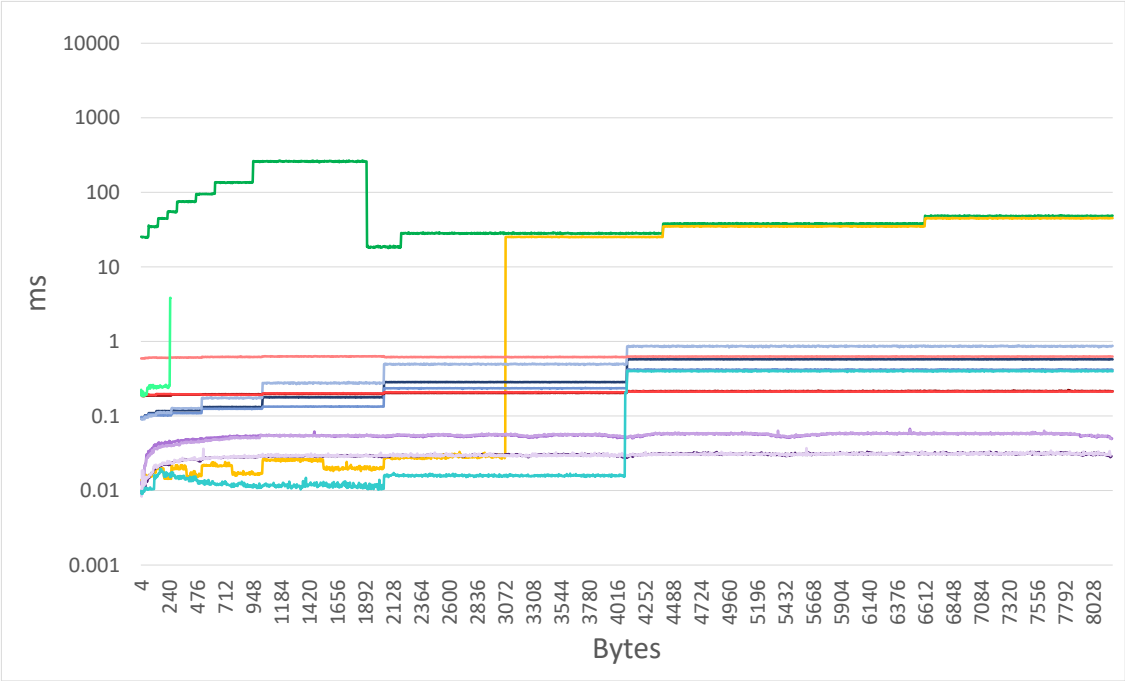
(d) Deallocation performance 10K - **median**



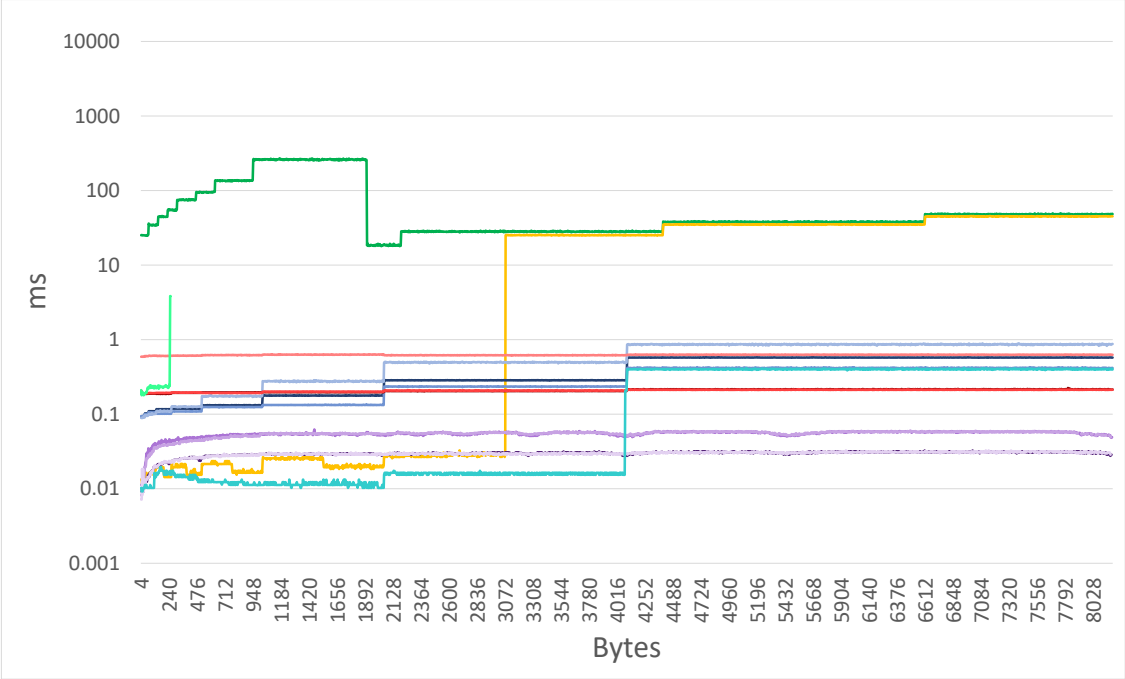
(e) Allocation performance 100K - **mean**



(f) Allocation performance 100K - **median**

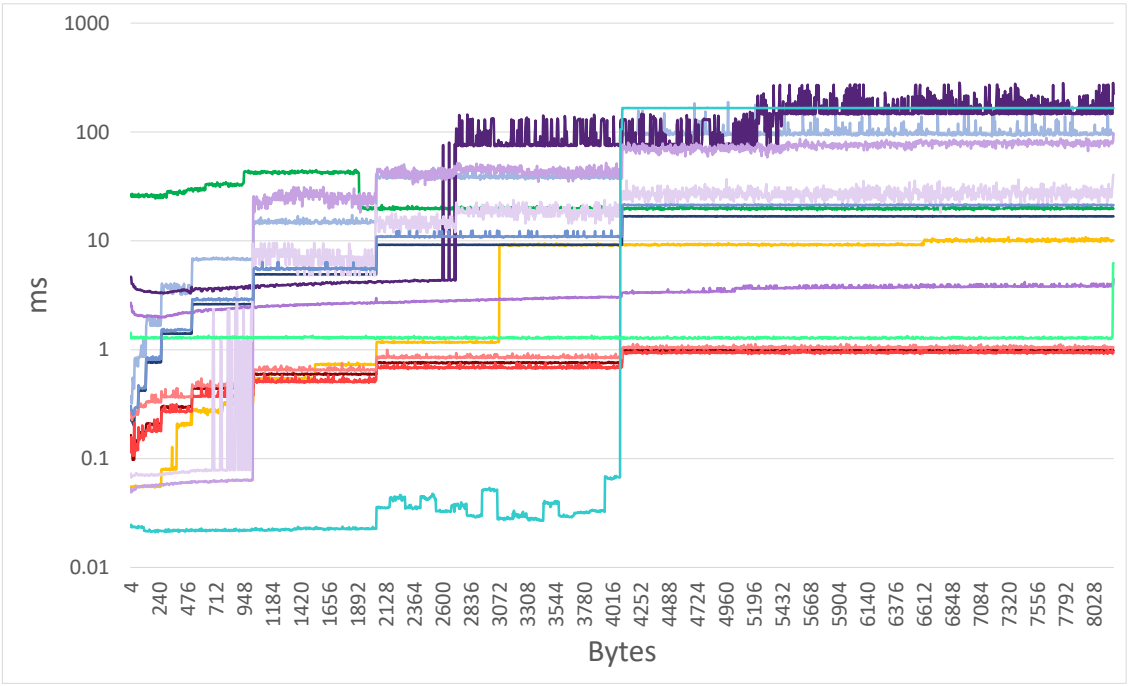


(g) Deallocation performance 100K - **mean**

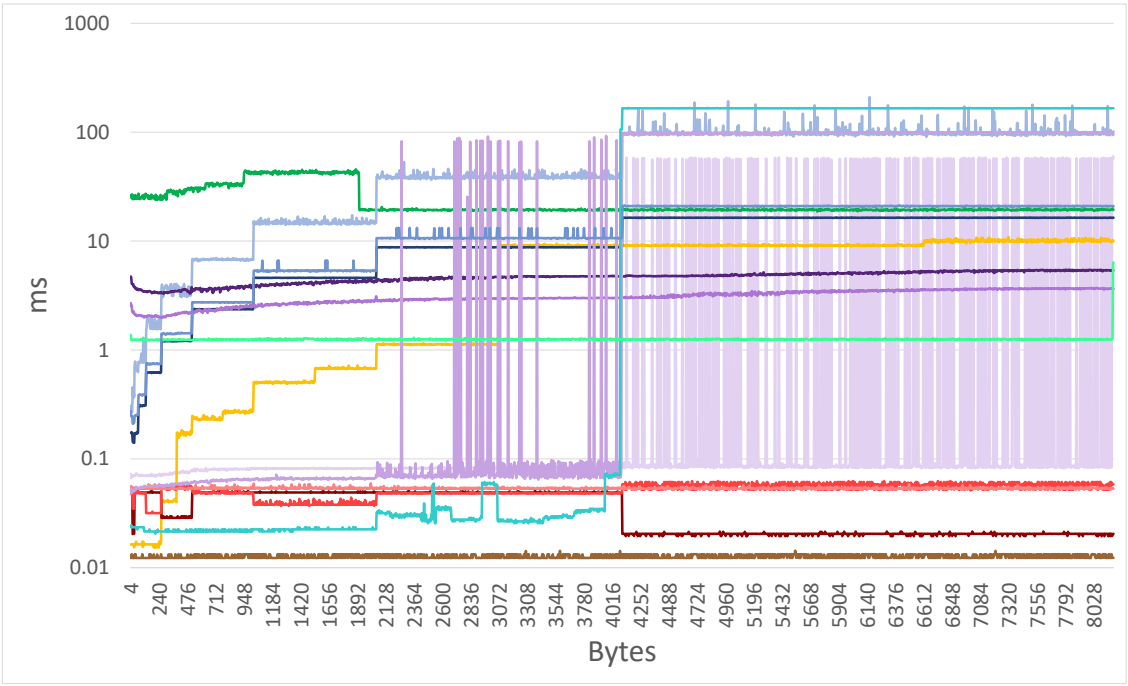


(h) Deallocation performance 100K - **median**

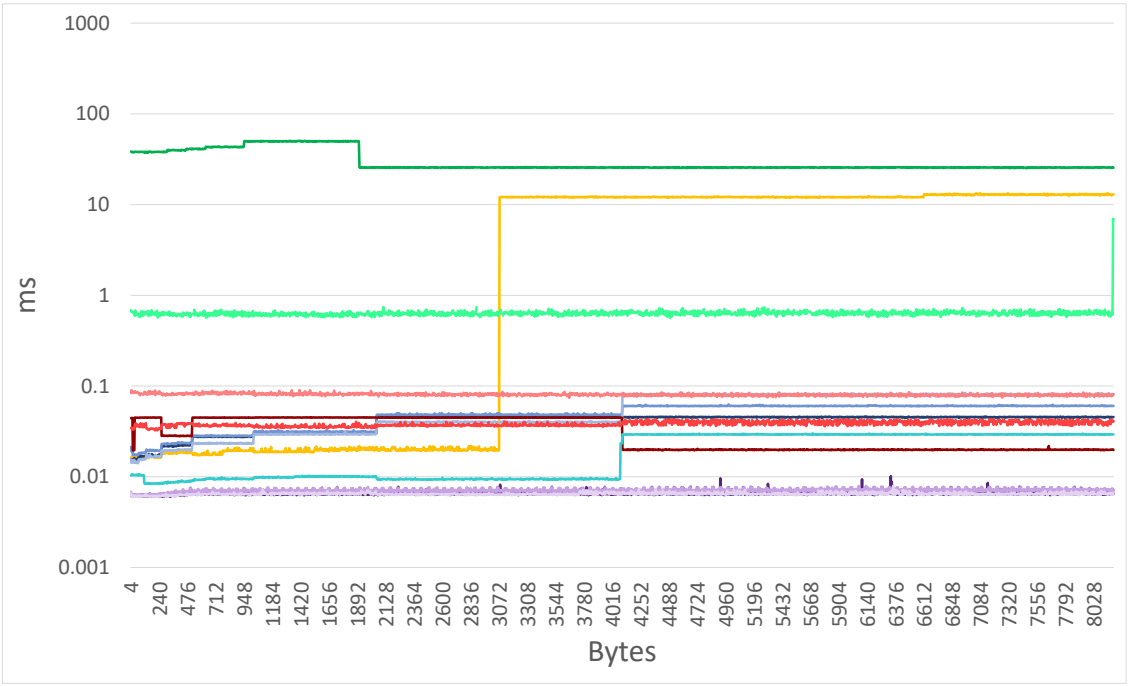
Figure B.2: Thread-based allocation and deallocation performance for 10.000 and 100.000 threads



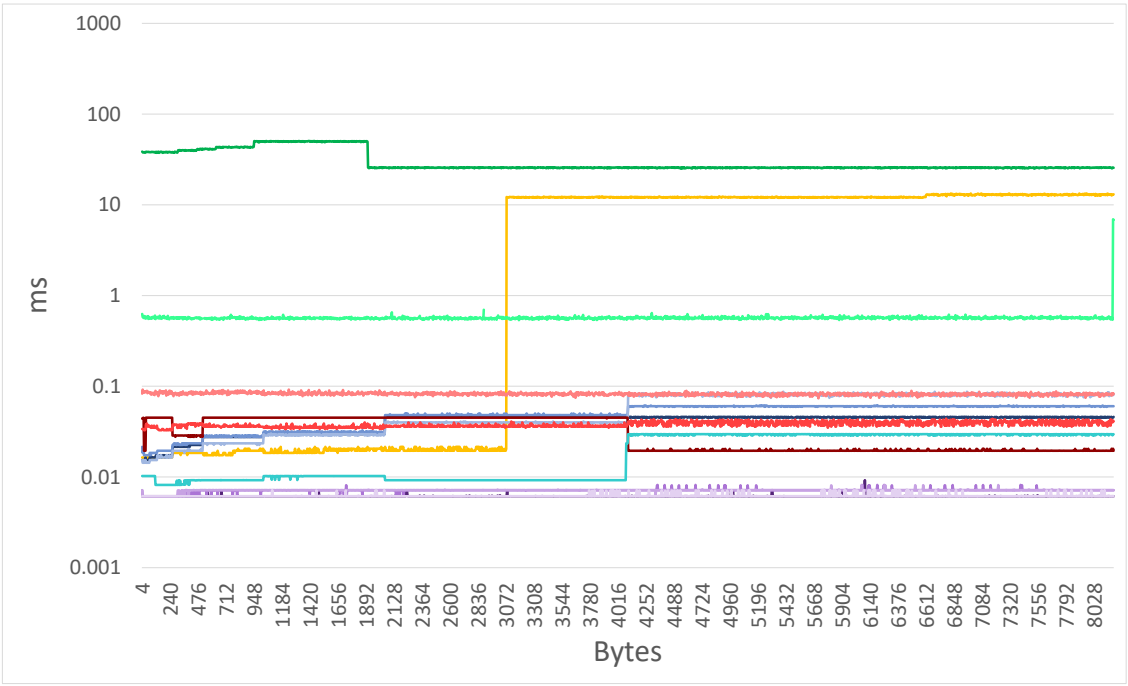
(a) Allocation performance 10K - **mean**



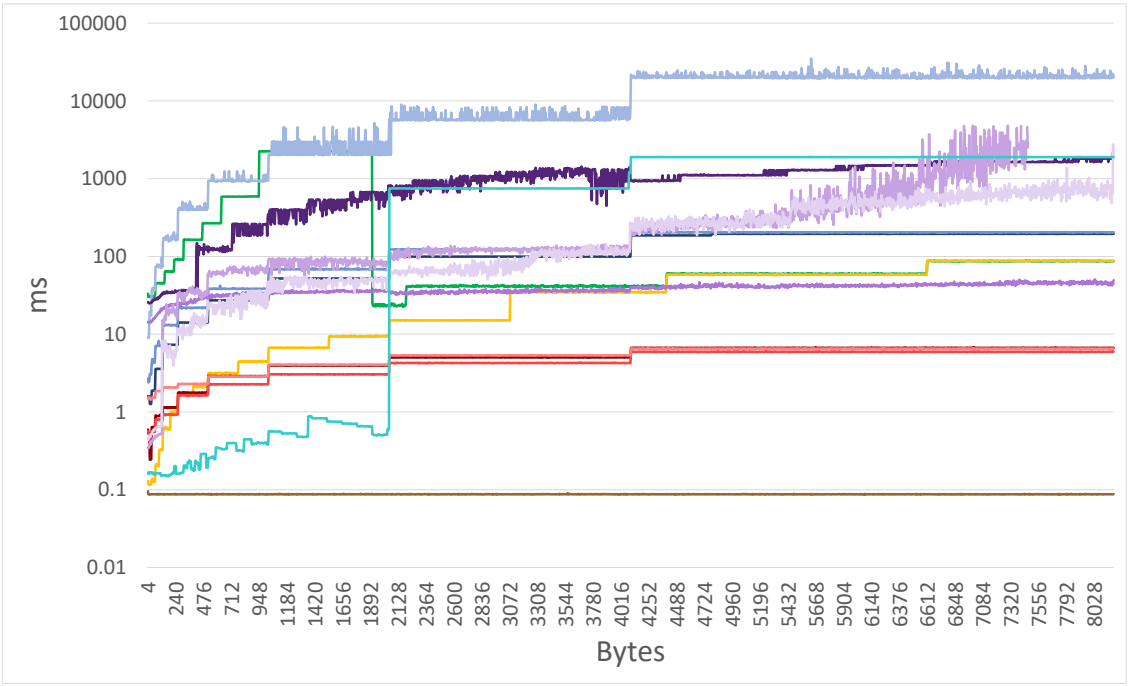
(b) Allocation performance 10K - **median**



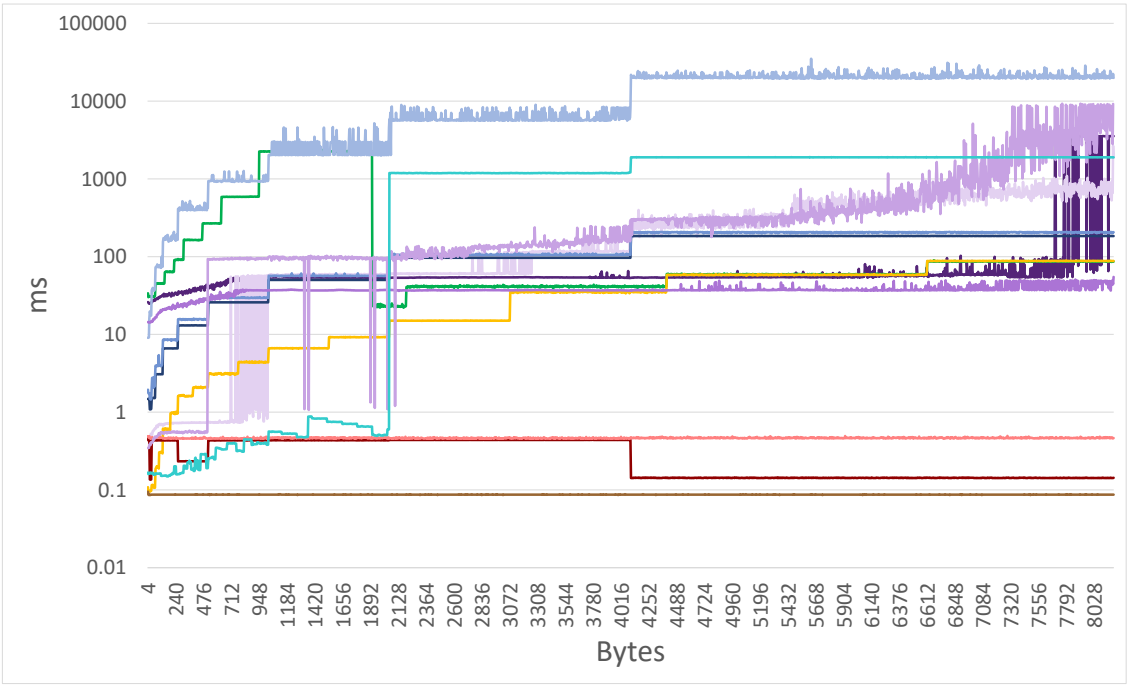
(c) Deallocation performance 10K - **mean**



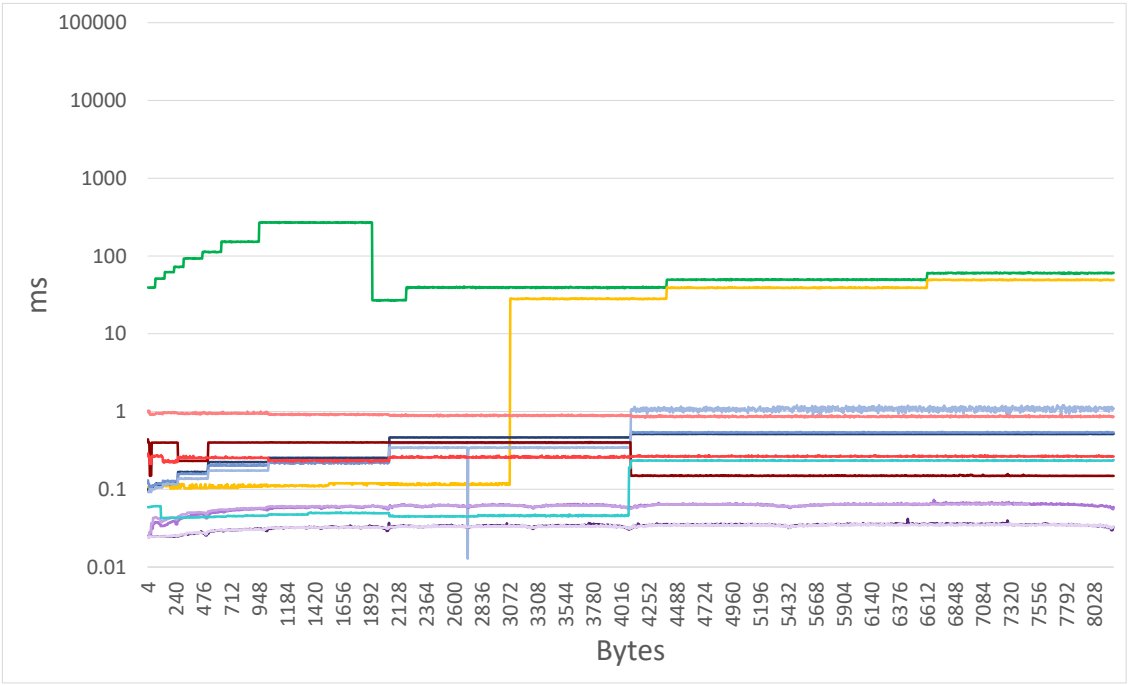
(d) Deallocation performance 10K - **median**



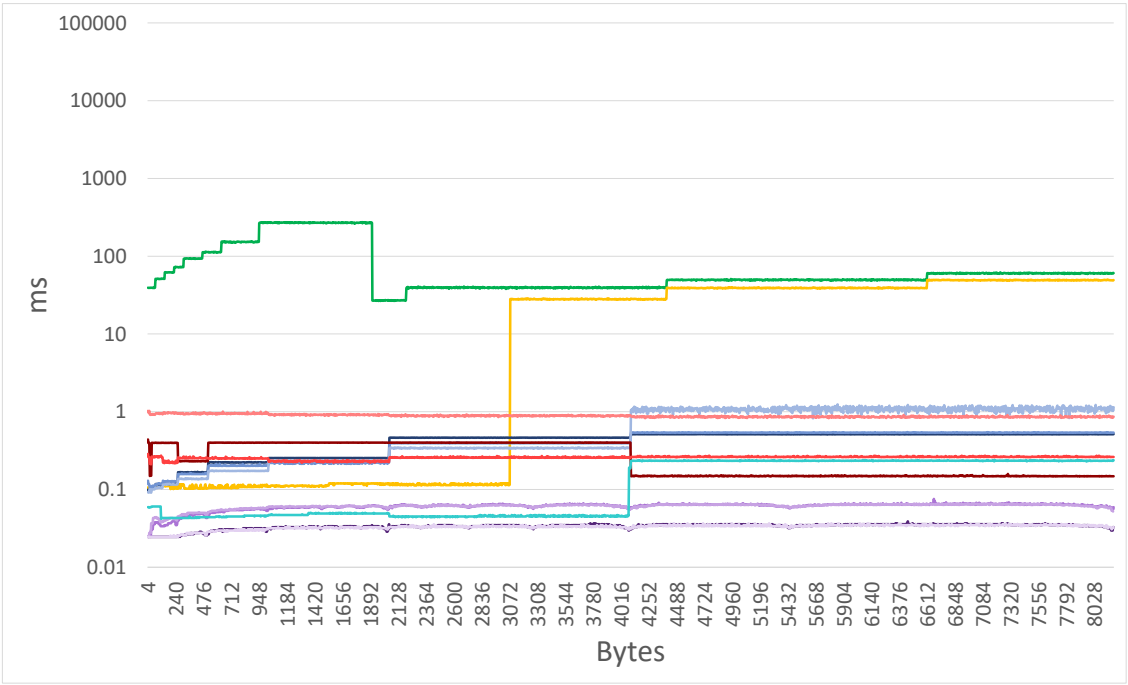
(e) Allocation performance 100K - **mean**



(f) Allocation performance 100K - **median**

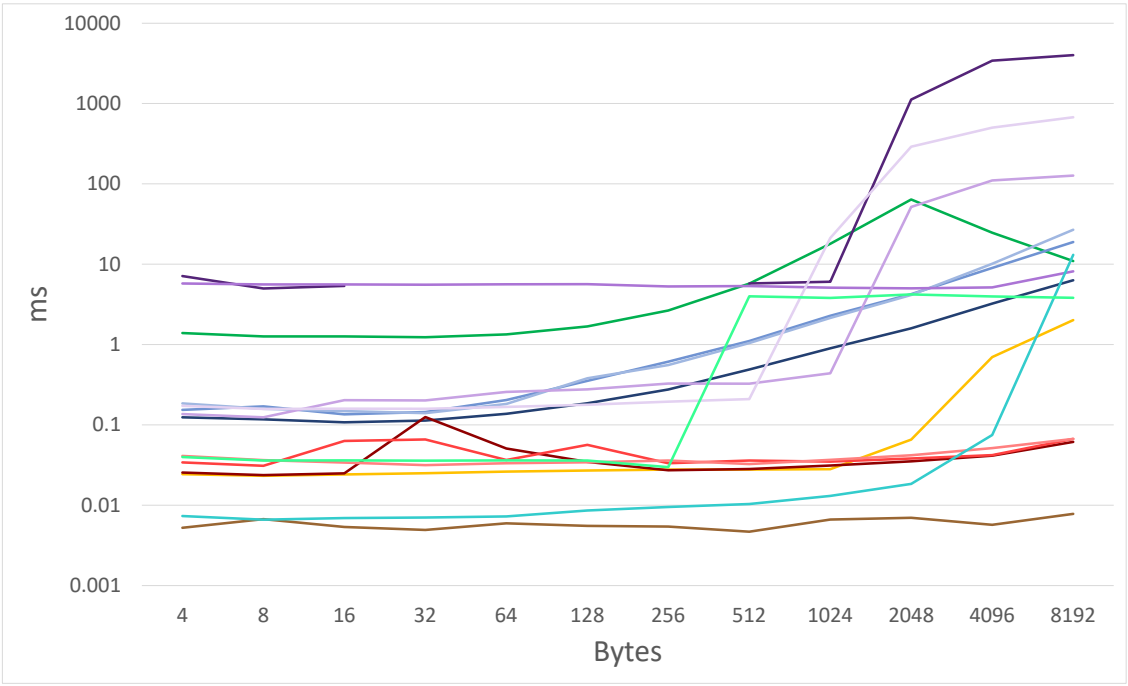


(g) Deallocation performance 100K - **mean**

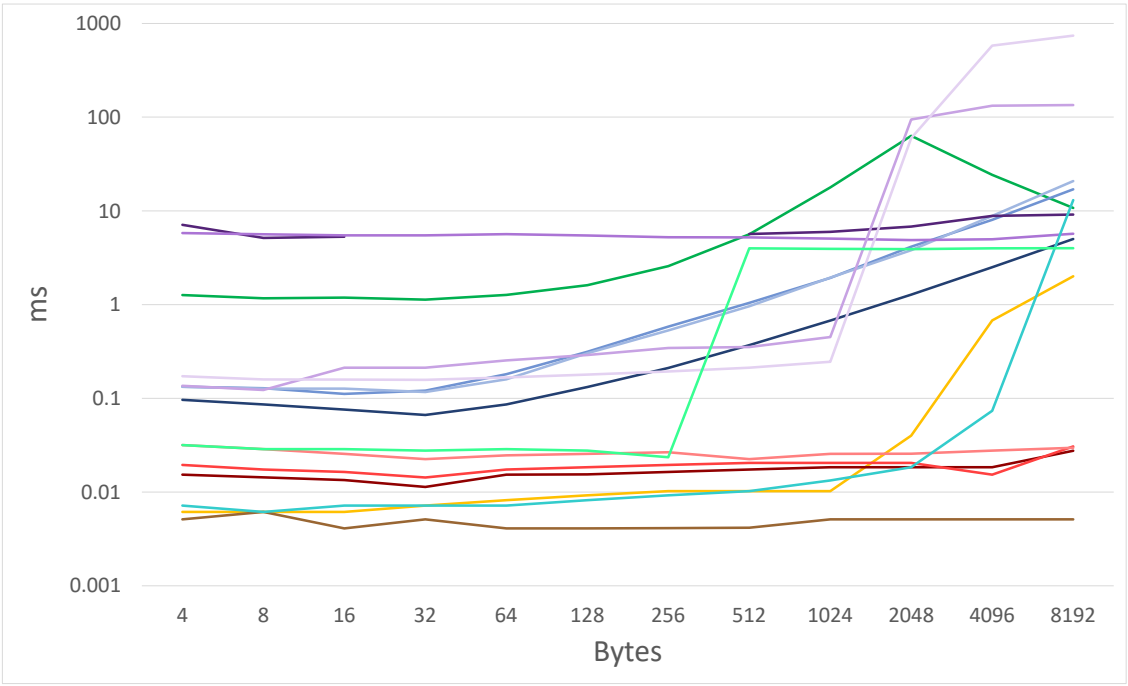


(h) Deallocation performance 100K - **median**

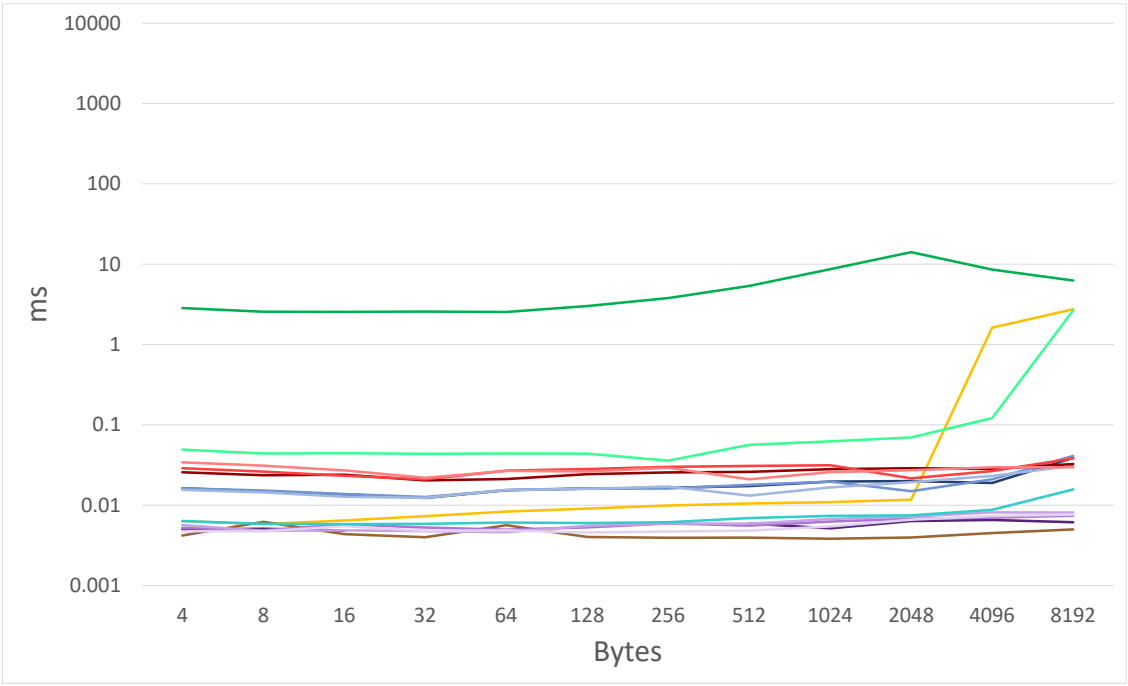
Figure B.3: Warp-based allocation and deallocation performance for 10.000 and 100.000 threads.



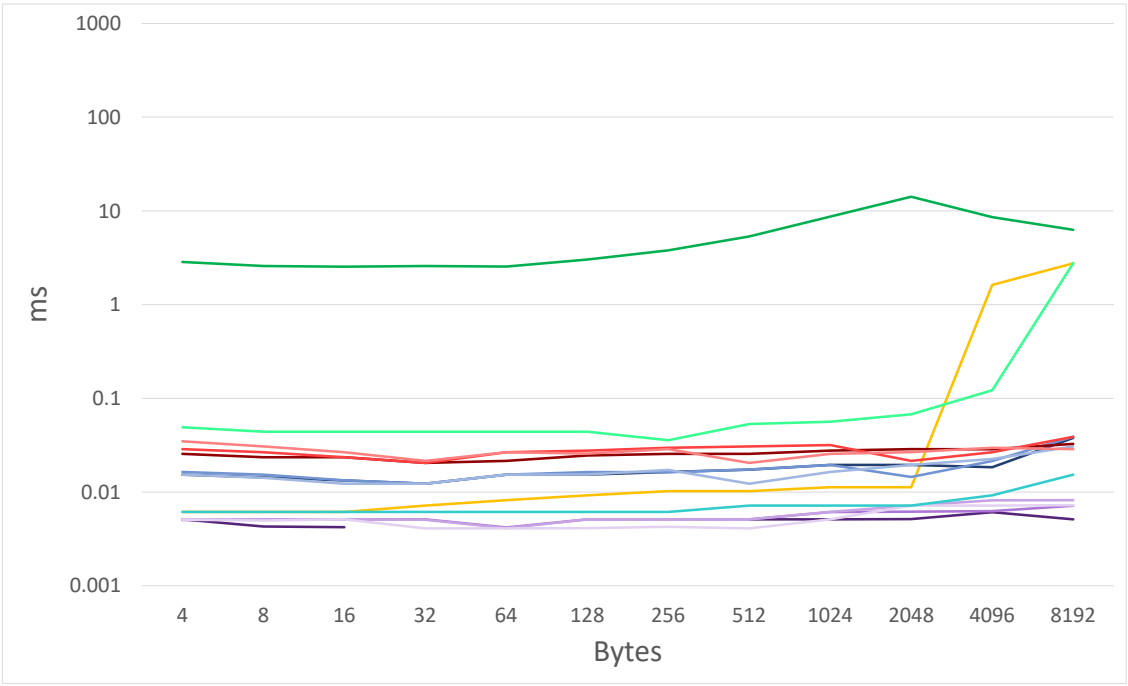
(a) Allocation performance 10K - **mean**



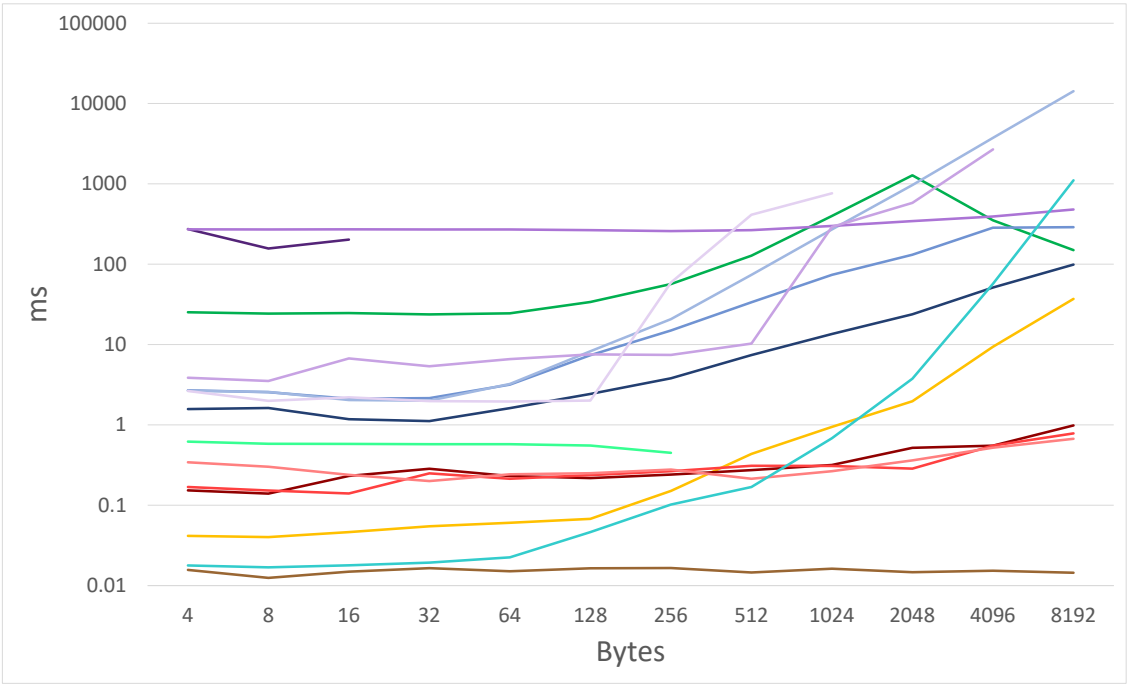
(b) Allocation performance 10K - **median**



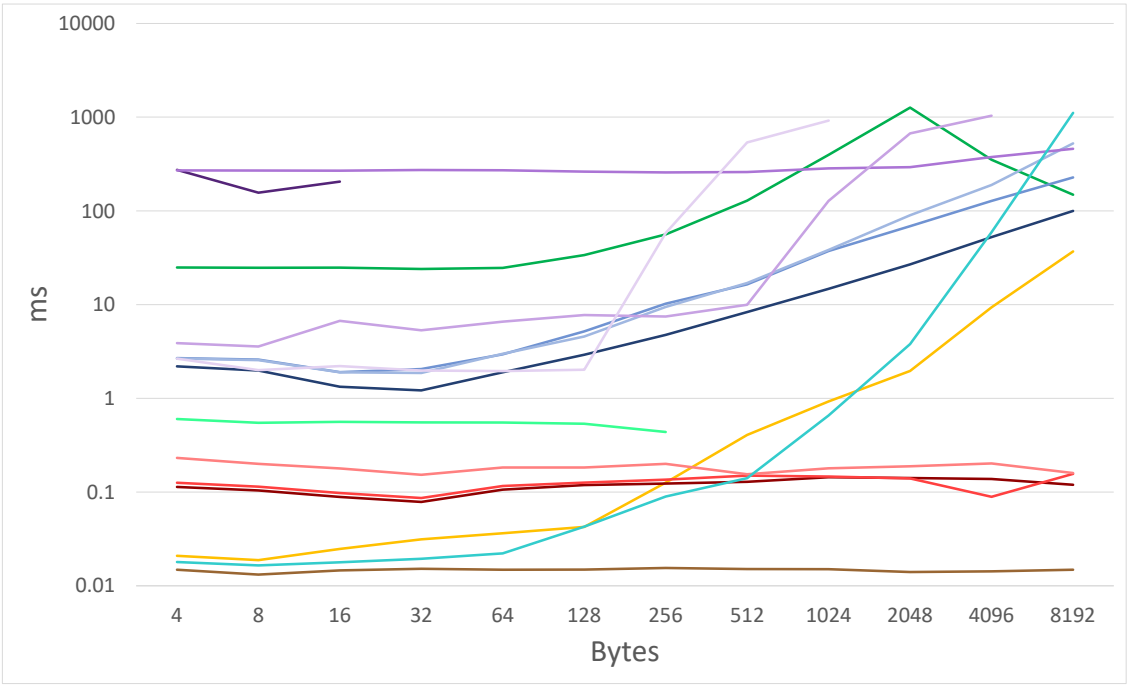
(c) Deallocation performance 10K - **mean**



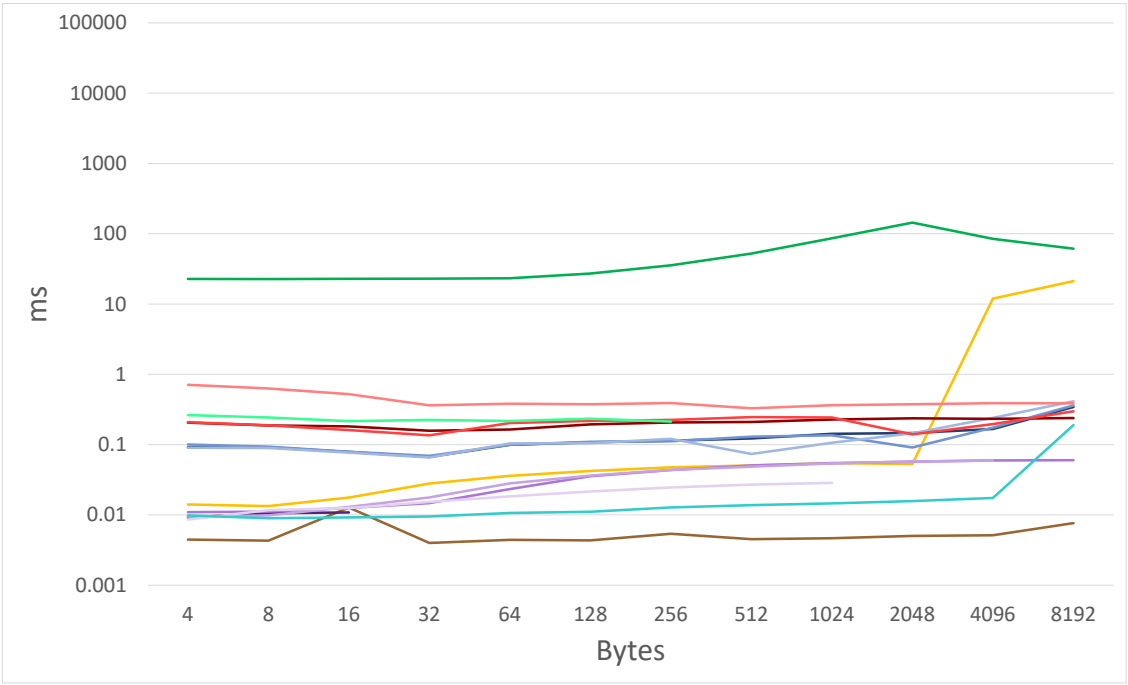
(d) Deallocation performance 10K - **median**



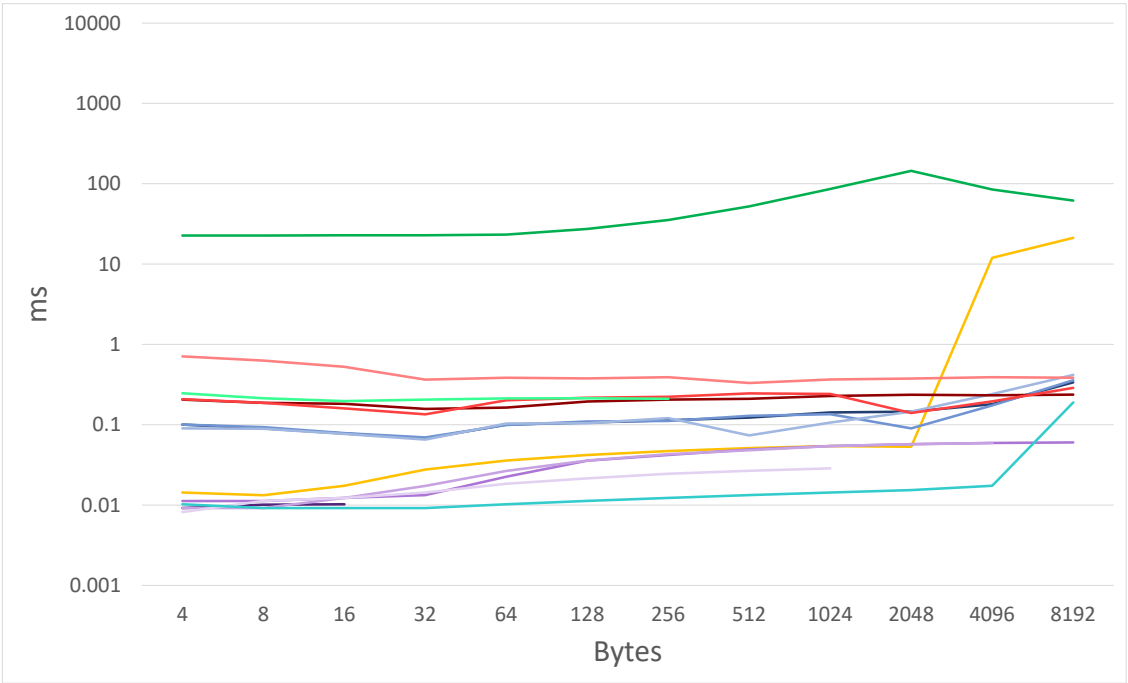
(e) Allocation performance 100K - **mean**



(f) Allocation performance 100K - **median**

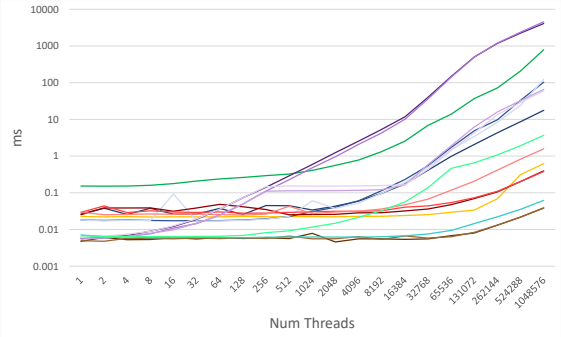


(g) Deallocation performance 100K - **mean**

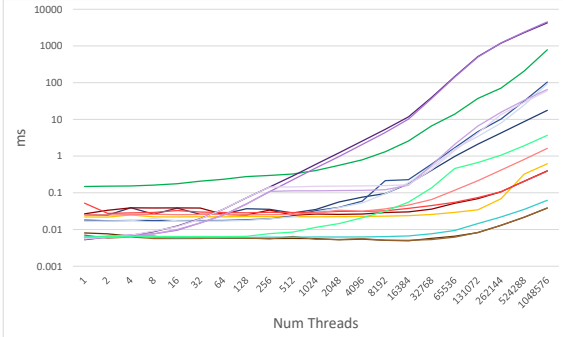


(h) Deallocation performance 100K - **median**

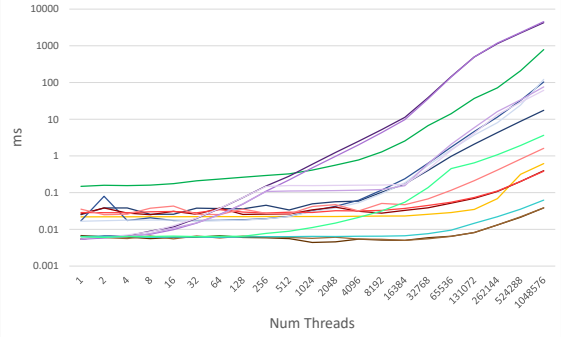
Figure B.4: Mixed allocation and deallocation performance for 10.000 and 100.000 threads



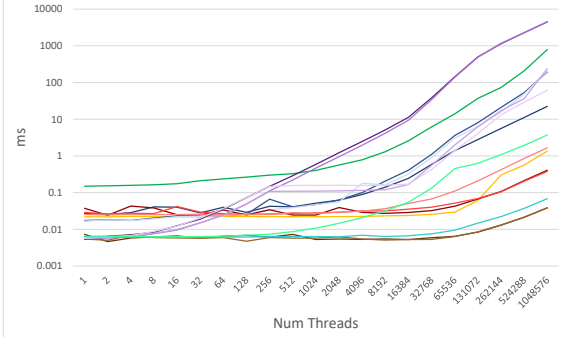
(a) Scaling 4 B



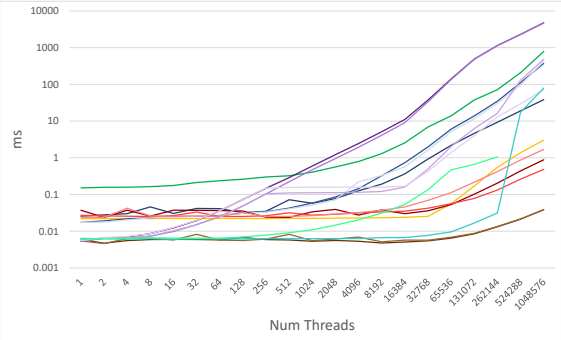
(b) Scaling 8 B



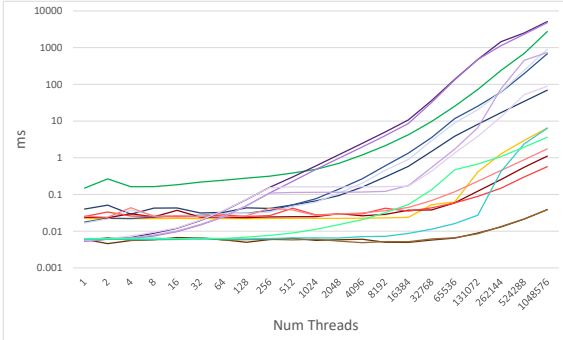
(c) Scaling 16 B



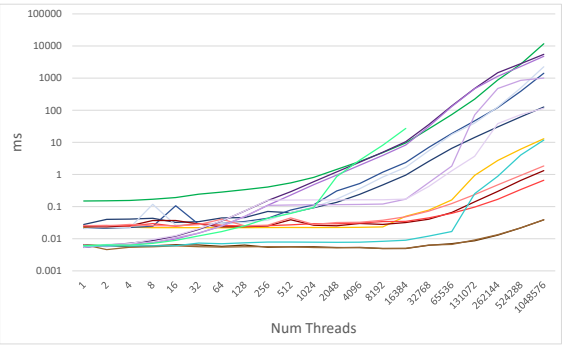
(d) Scaling 32 B



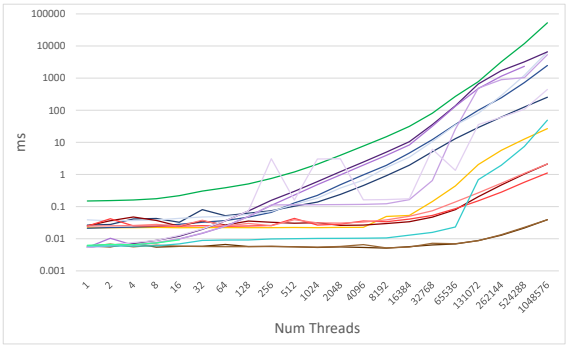
(e) Scaling 64 B



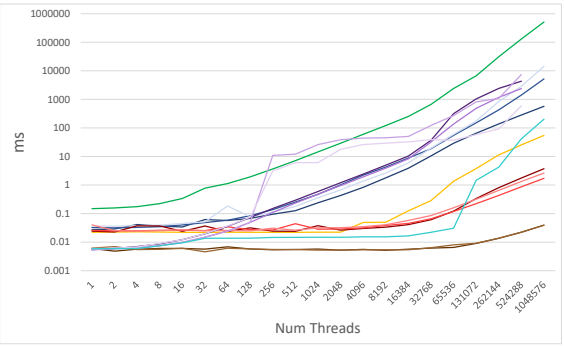
(f) Scaling 128 B



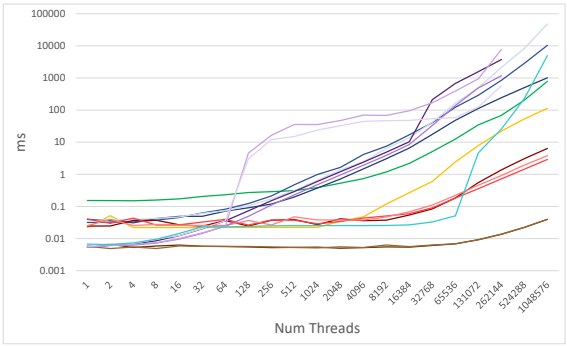
(g) Scaling 256 B



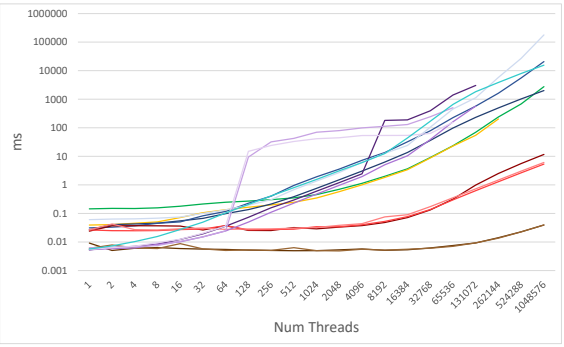
(h) Scaling 512 B



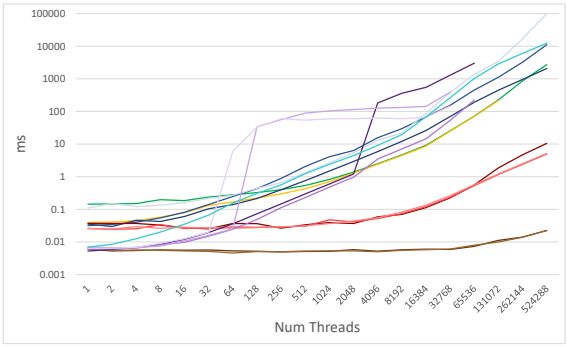
(i) Scaling 1024 B



(j) Scaling 2048 B

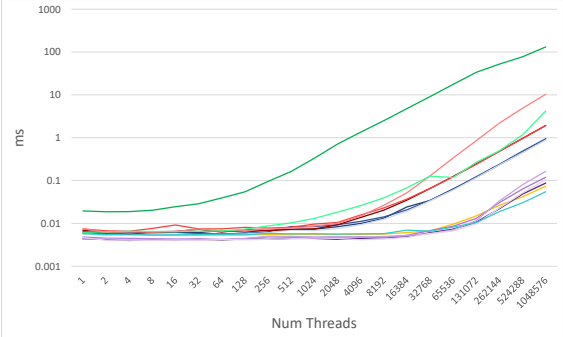


(k) Scaling 4096 B

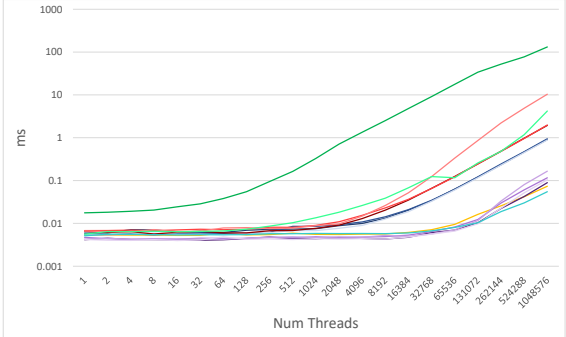


(l) Scaling 8192 B

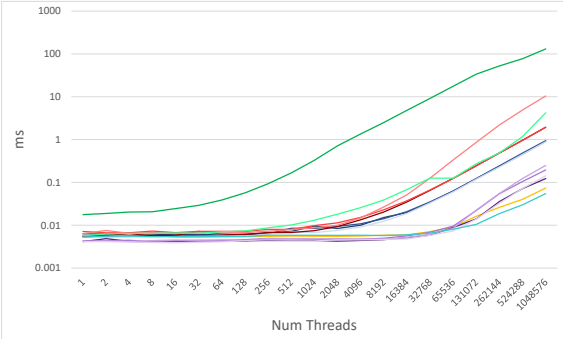
Figure B.5: Scaling performance for allocation in range 4 B–8192 B



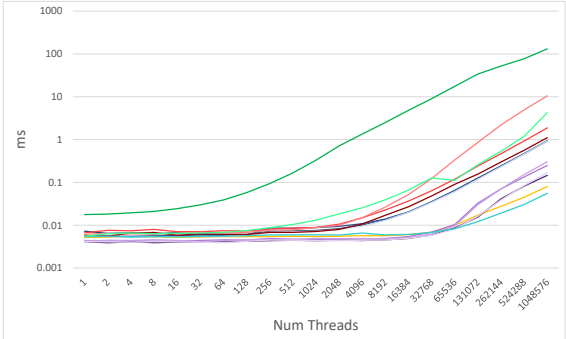
(a) Scaling 4 B



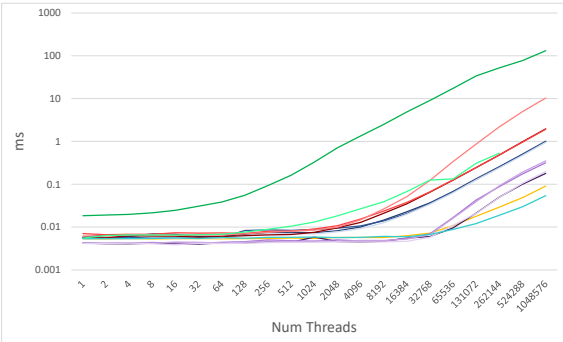
(b) Scaling 8 B



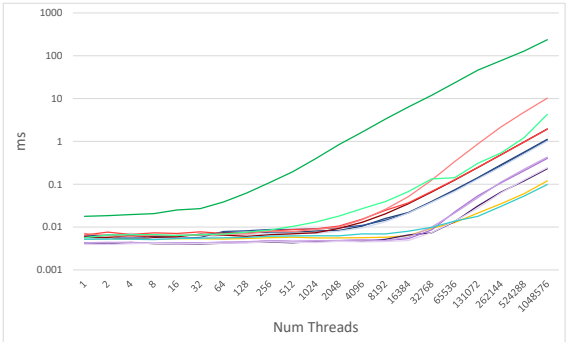
(c) Scaling 16 B



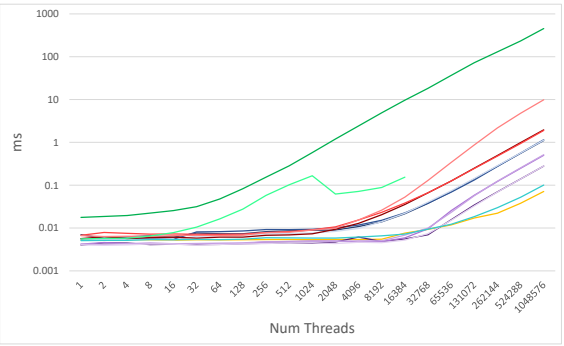
(d) Scaling 32 B



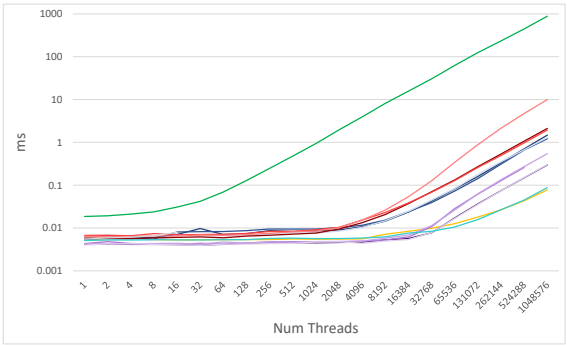
(e) Scaling 64 B



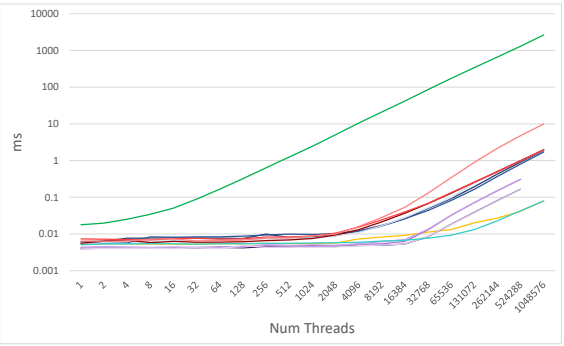
(f) Scaling 128 B



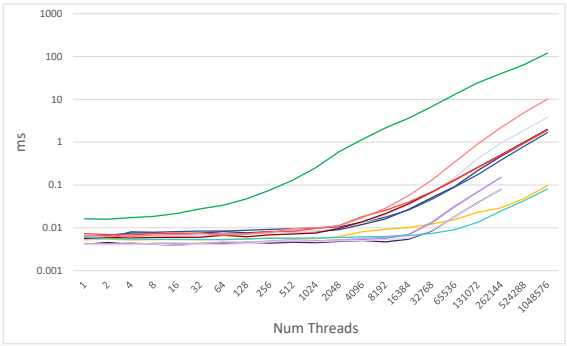
(g) Scaling 256 B



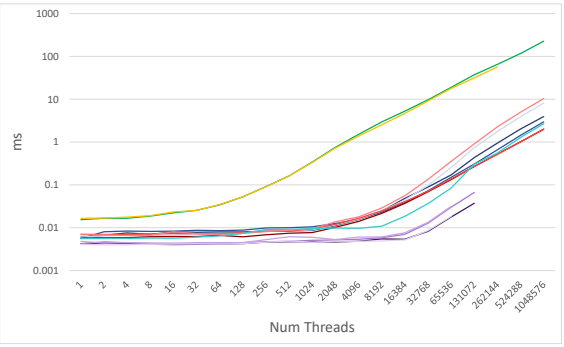
(h) Scaling 512 B



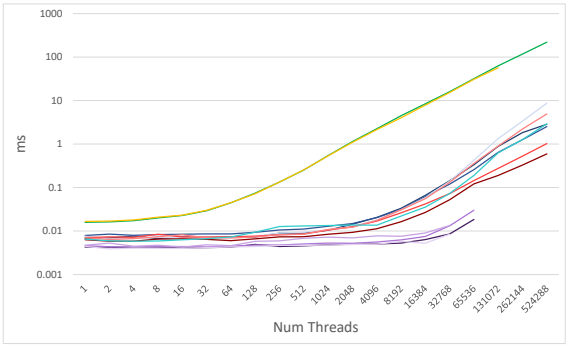
(i) Scaling 1024 B



(j) Scaling 2048 B

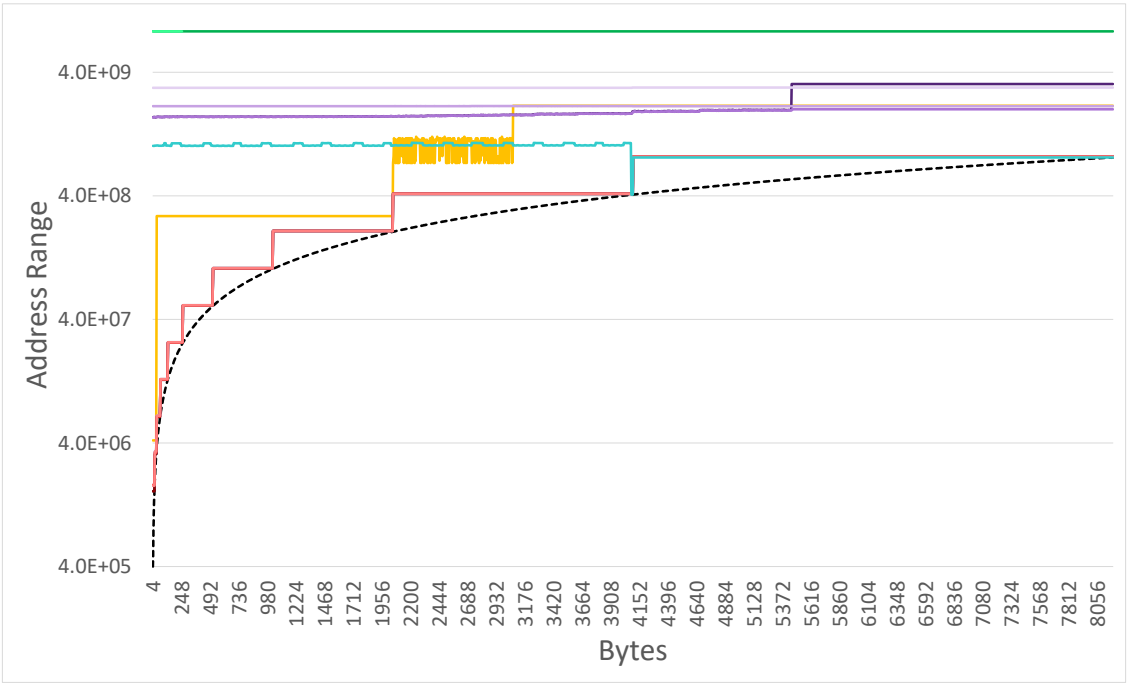


(k) Scaling 4096 B

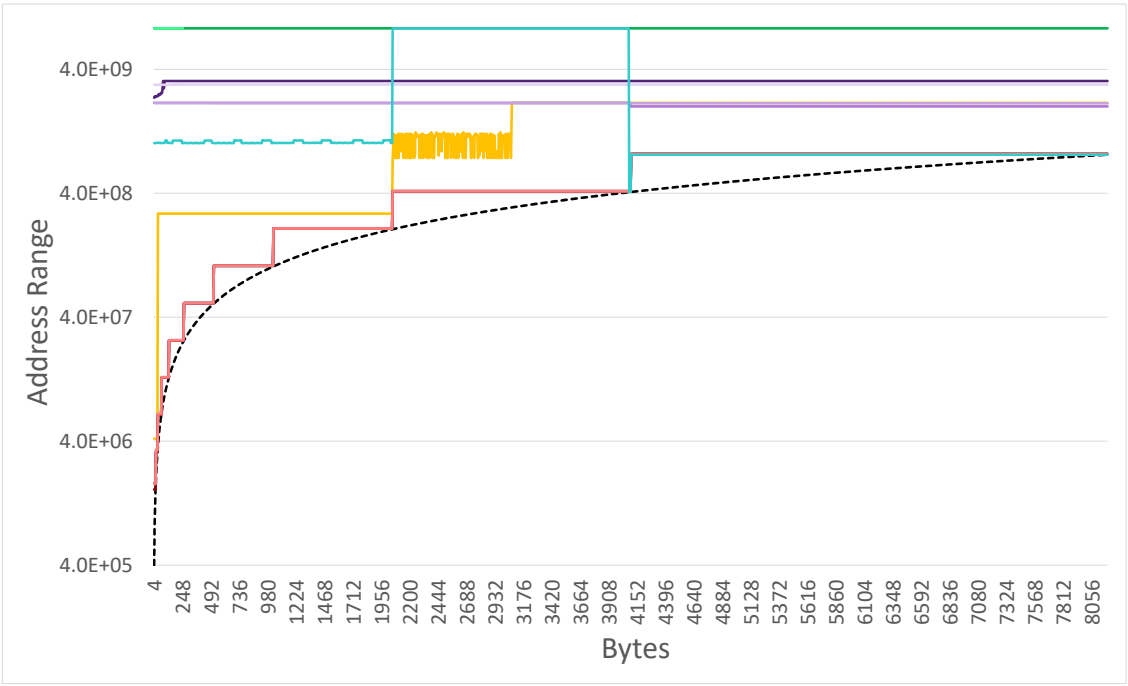


(l) Scaling 8192 B

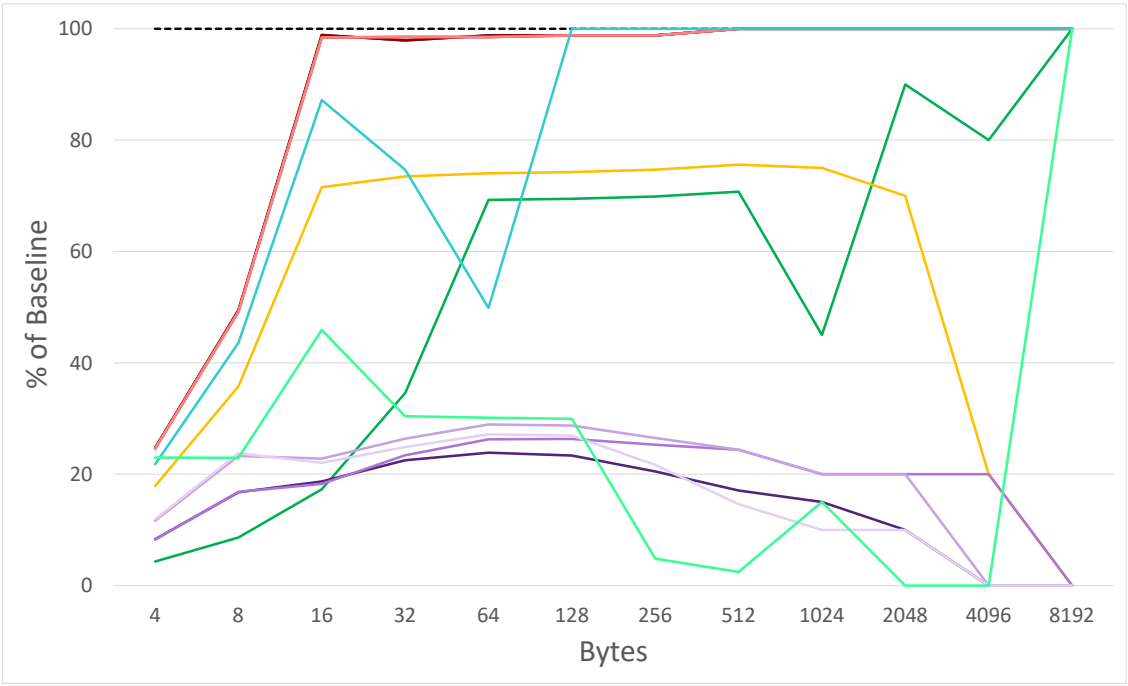
Figure B.6: Scaling performance for deallocation in range 4 B–8192 B



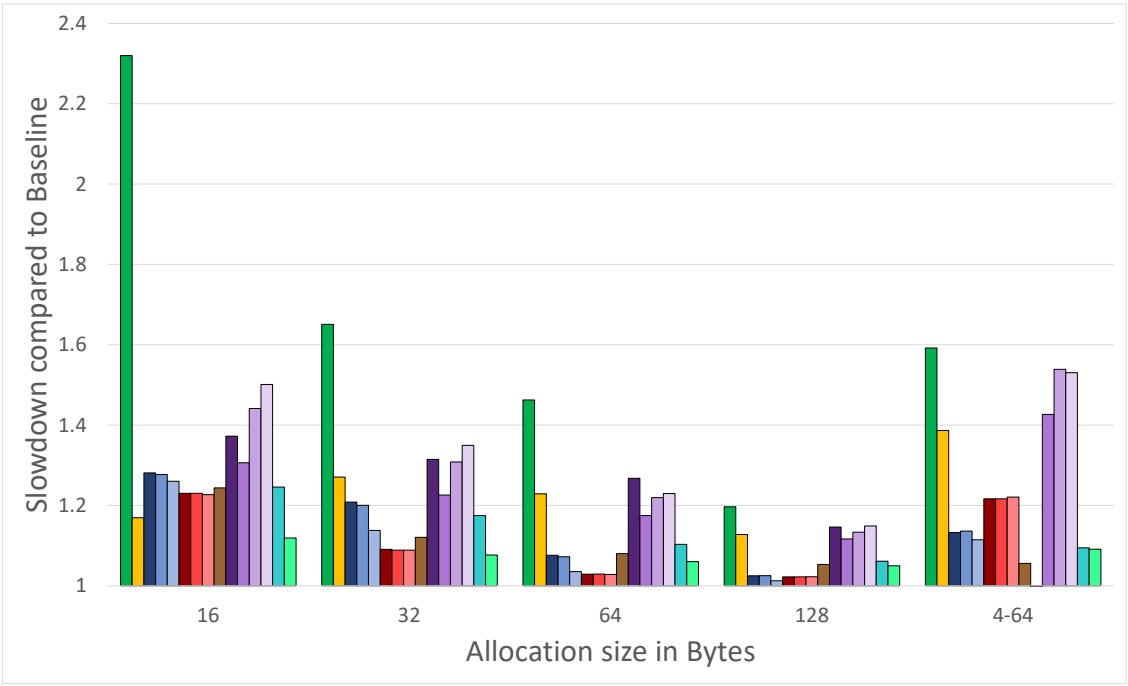
(a) Fragmentation after one round



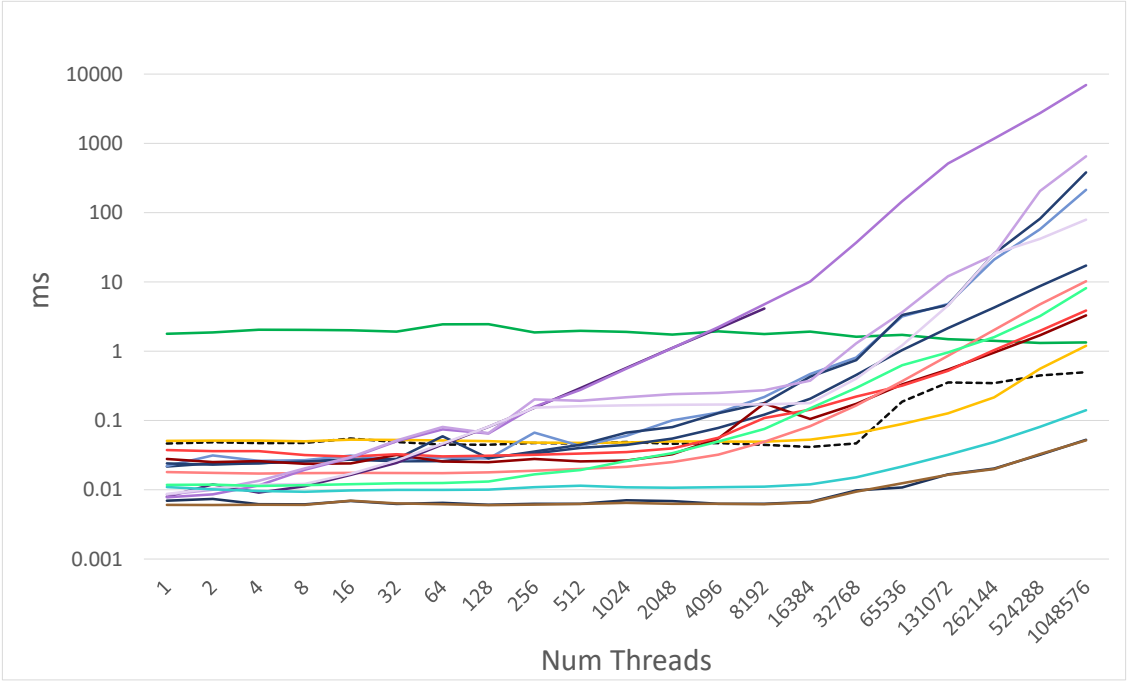
(b) Static fragmentation after 100 rounds



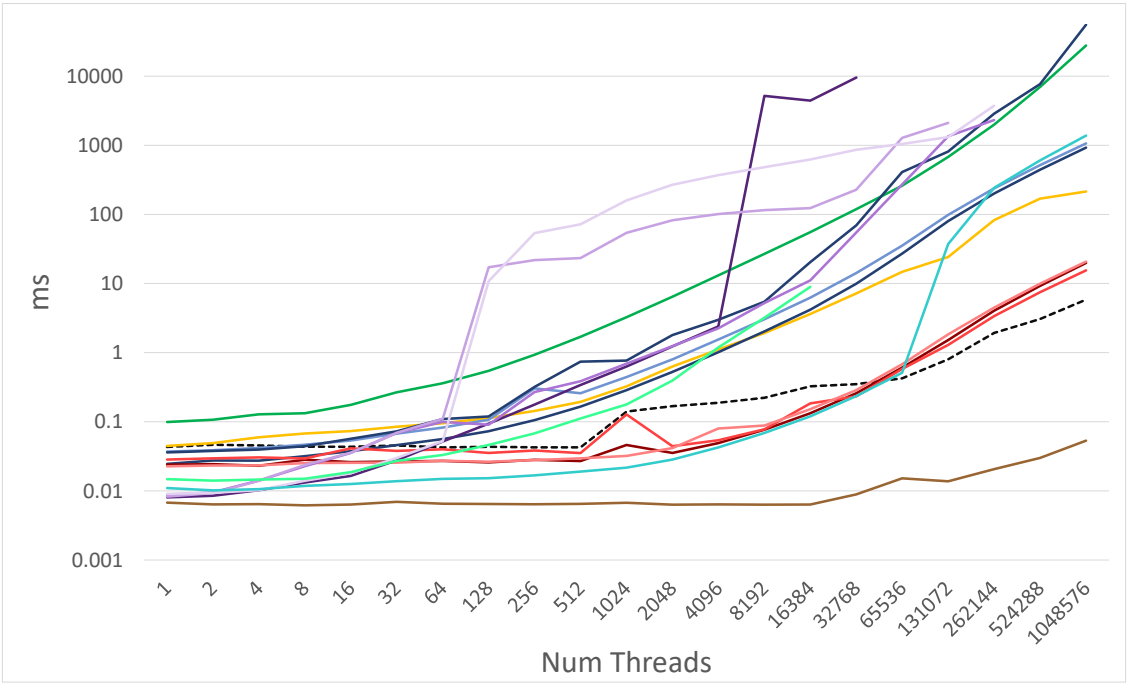
(c) Out-of-Memory



(d) Write performance compared to Baseline

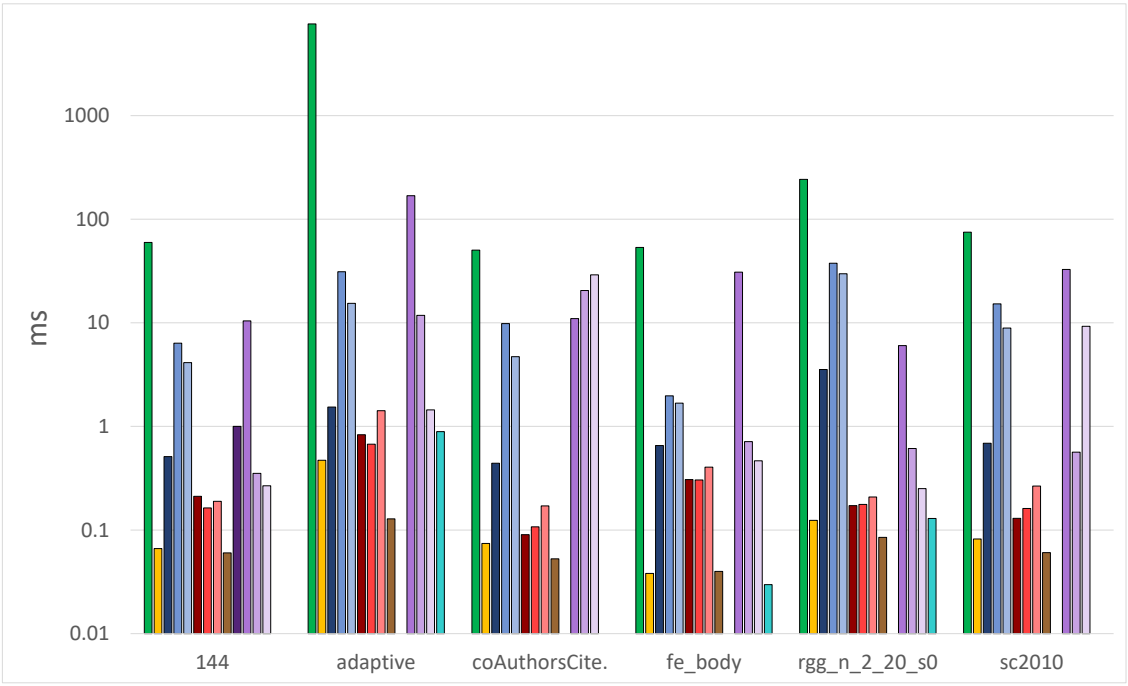


(e) Synthetic workload for range 4 B–64 B

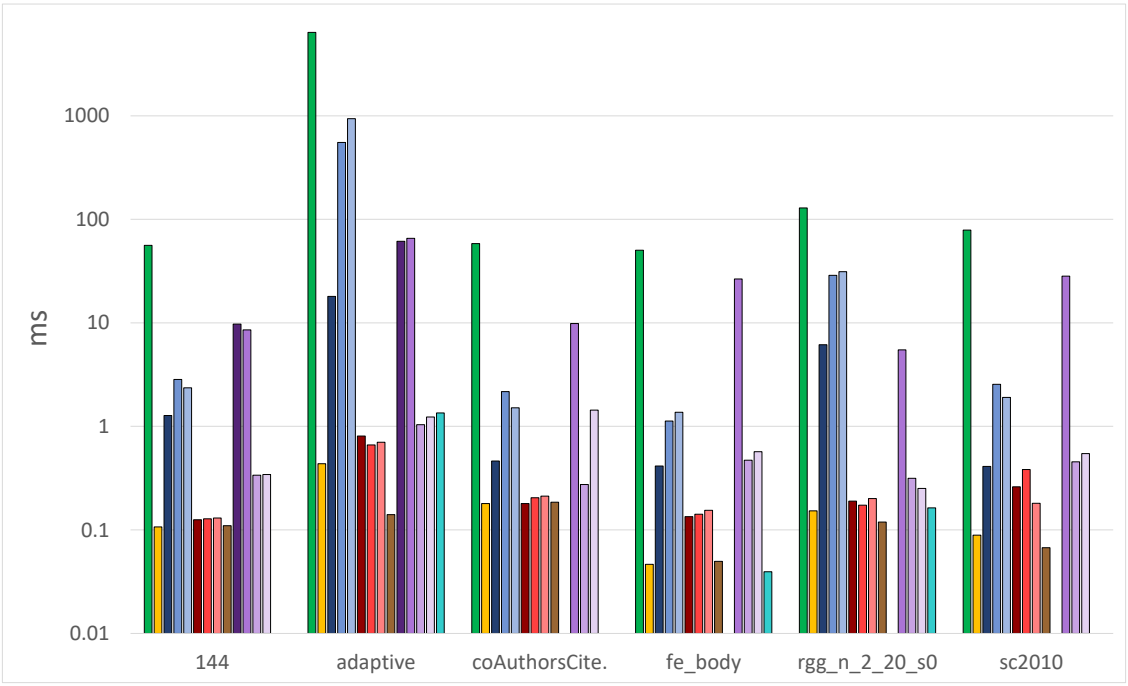


(f) Synthetic workload for range 4 B–4096 B

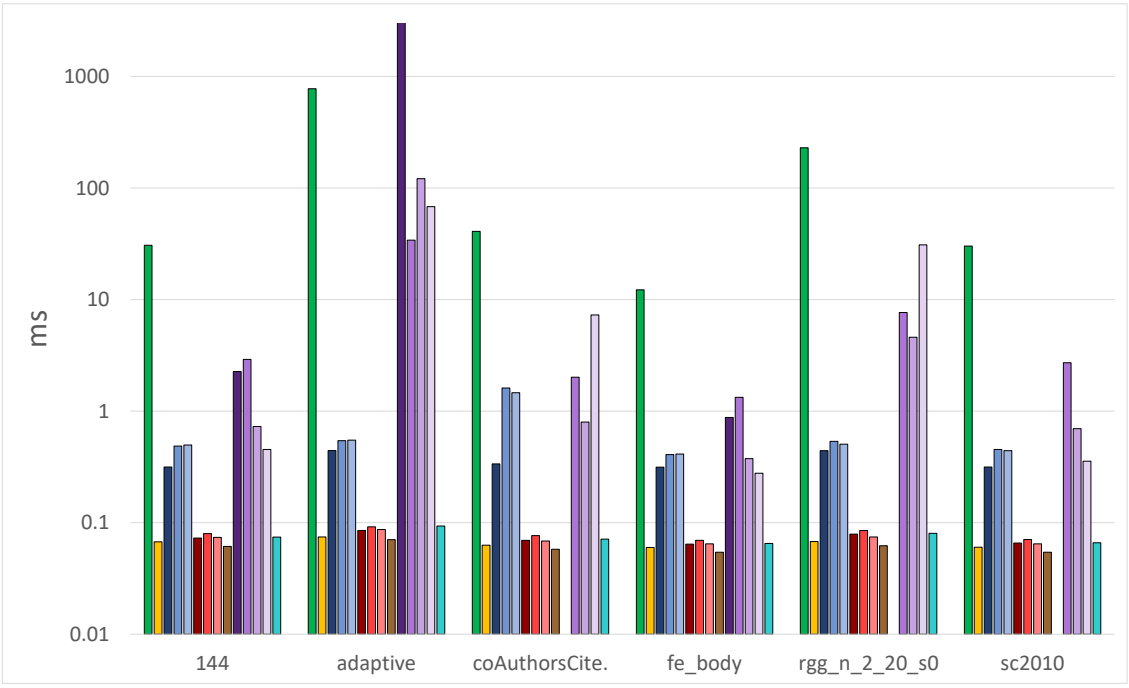
Figure B.7: Fragmentation after one round and static fragmentation after 100 rounds of allocating a specific size and out-of-memory testcase and two synthetic workload test for 4 B–64 B and 4 B–4096 B of work generated per thread.



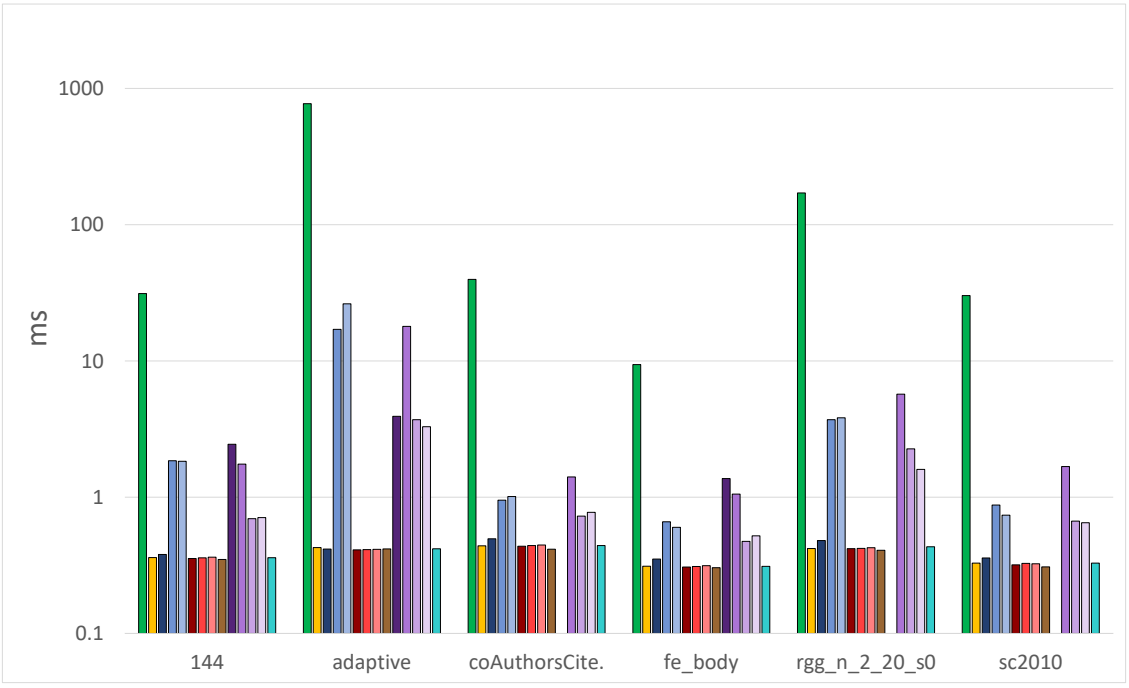
(a) Edge Insertion uniform



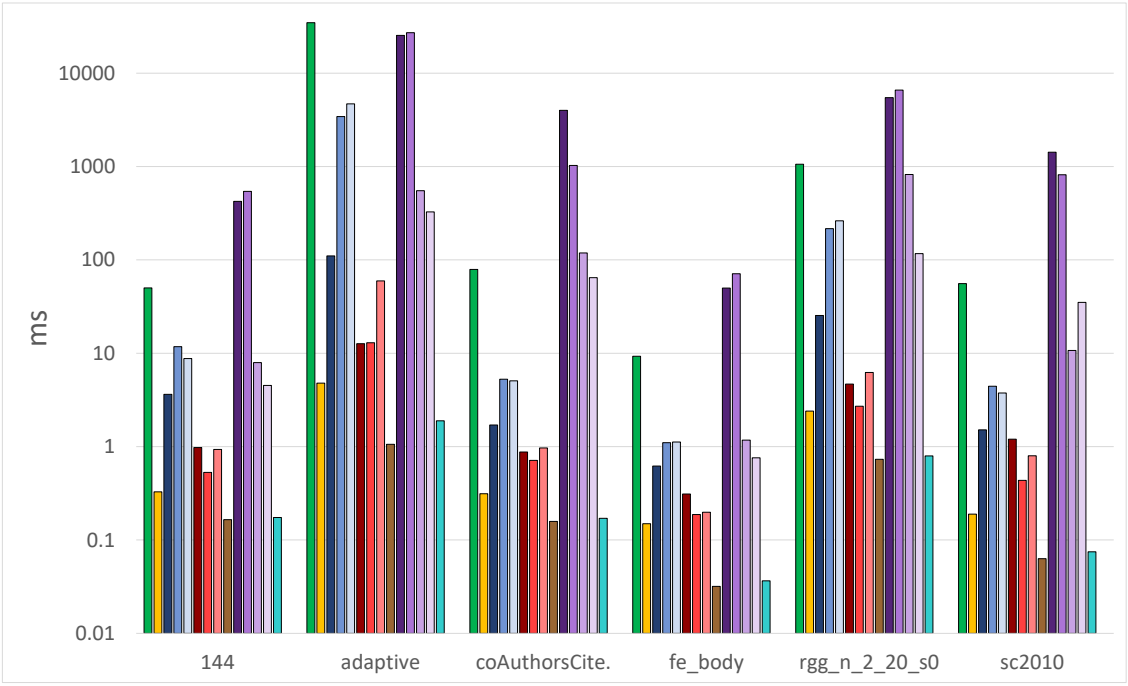
(b) Edge Deletion uniform



(c) Edge Insertion focused on range of 1000 source vertices



(d) Edge Deletion focused on range of 1000 source vertices



(e) Graph Initialization

Figure B.8: Graph performance: Initialization as well as uniform updates and updates focused on range of 1000 source vertices, edge update batch size 100.000

Bibliography

[1] A. V. Adinetz and D. Pleiter. Halloc: A High-Throughput Dynamic Memory Allocator for GPGPU architectures. In *GPU Technology Conference (GTC'14)*, volume 152, 2014. (page 7, 25)

[2] S. Ashkiani, M. Farach-Colton, and J. D. Owens. A Dynamic Hash Table for the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 419–429. IEEE, 2018, https://doi.org/10.1109/IPDPS.2018.00052. (page 6, 15)

[3] M. A. Awad, S. Ashkiani, S. D. Porumbescu, and J. D. Owens. Dynamic Graphs on the GPU. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 739–748, 2020, https://doi.org/10.1109/IPDPS47924.2020.00081. (page 4, 15, 67)

[4] M. A. Awad, S. Ashkiani, R. Johnson, M. Farach-Colton, and J. D. Owens. Engineering a High-Performance GPU B-tree. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 145–157, 2019, https://doi.org/10.1145/3293883.3295706. (page 15)

[5] D. A. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82, Atlanta, Georgia, USA, 2014. AMS, https://doi.org/10.1007/978-1-4939-7131-2_23. (page 40, 41, 59, 80, 119, 127)

[6] D. A. Bader, J. Berry, A. Amos-Binks, D. Chavarría-Miranda, C. Hastings, K. Madduri, and S. C. Poulos. Stinger: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation. 2009. (page 10)

[7] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. *Graph Partitioning and Graph Clustering*, volume 588. American Mathematical Society Providence, RI, 2013. (page 31)

[8] M. A. Bender and H. Hu. An Adaptive Packed-Memory Array. volume 32, page 26–69, New York, NY, USA, November 2007. Association for Computing Machinery, https://doi.org/10.1145/1292609.1292616. (page 11)

[9] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, page 117–128, New York, NY, USA, 2000. Association for Computing Machinery, https://doi.org/10.1145/378993.379232. (page 2)

[10] J. Bonwick and J. Adams. Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, page 15–33, USA, 2001. USENIX Association. (page 2)

[11] A. Brahmakshatriya, Y. Zhang, C. Hong, S. Kamil, J. Shun, and S. Amarasinghe. Compiling Graph Applications for GPUs with GraphIt. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 248–261, 2021, https://doi.org/10.1109/CGO51591.2021.9370321. (page 10)

[12] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. volume 30, pages 107–117. Elsevier, 1998, https://doi.org/10.1016/S0169-7552(98)00110-X. (page 31, 71, 75)

[13] F. Busato, O. Green, N. Bombieri, and D. A. Bader. Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs. In *2018 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018, https://doi.org/10.1109/HPEC.2018.8547541. (page 4, 13, 119)

[14] S. Che. GasCL: A Vertex-Centric Graph Model for GPUs. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2014, https://doi.org/10.1109/HPEC.2014.7040962. (page 2, 9)

[15] S. Che, B. M. Beckmann, and S. K. Reinhardt. Belred: Constructing GPGPU Graph Applications with Software Building Blocks. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2014, https://doi.org/10.1109/HPEC.2014.7040961. (page 2, 9)

[16] Chien-Hua Shann, Ting-Lu Huang, and Cheng Chen. A practical nonblocking queue algorithm using compare-and-swap. In *Proceedings Seventh International Conference on Parallel and Distributed Systems (Cat. No.PR00568)*, pages 470–475, Iwate, Japan, Japan, July 2000. IEEE, https://doi.org/10.1109/ICPADS.2000.857731. (page 30)

[17] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient Parallel GPU Methods for Single-Source Shortest Paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 349–359. IEEE, 2014, https://doi.org/10.1109/IPDPS.2014.45. (page 31)

[18] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. Stinger: High Performance Data Structure for Streaming Graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5. IEEE, 2012, https://doi.org/10.1109/HPEC.2012.6408680. (page 10)

[19] J. Evans. A Scalable Concurrent malloc (3) Implementation for FreeBSD. In *Proc. of the BSDCan Conference, Ottawa, Canada*, 2006. (page 2)

[20] I. Gelado and M. Garland. Throughput-oriented GPU Memory Allocation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 27–37, New York, NY, USA, 2019. ACM, `http://doi.acm.org/10.1145/3293883.3295727`. (page 6, 7, 29, 82, 84)

[21] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. volume 5, pages 164–189, New York, NY, USA, April 1983. ACM, `http://doi.acm.org/10.1145/69624.357206`. (page 30)

[22] O. Green and D. A. Bader. cuSTINGER: Supporting Dynamic Graph Algorithms for GPUs. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016, `https://doi.org/10.1109/HPEC.2016.7761622`. (page 4, 10, 48, 119)

[23] O. Green, P. Yalamanchili, and L.-M. Munguía. Fast Triangle Counting on the GPU. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–8, 2014, `https://dl.acm.org/doi/pdf/10.5555/2688283.2688284`. (page 31, 71, 72)

[24] M. Hoffman, O. Shalev, and N. Shavit. The Baskets Queue. In *Principles of Distributed Systems*, pages 401–414, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg, `https://doi.org/10.1007/978-3-540-77096-1_29`. (page 30)

[25] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W. Hwu. XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 1134–1139, Bradford, UK, June 2010. IEEE, `https://doi.org/10.1109/CIT.2010.206`. (page 7, 18)

[26] B. Kerbl, M. Kenzel, J. H. Mueller, D. Schmalstieg, and M. Steinberger. The Broker Queue: A Fast, Linearizable FIFO Queue for Fine-Granular Work Distribution on the GPU. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, page 76–85, New York, NY, USA, 2018. Association for Computing Machinery, `https://doi.org/10.1145/3205289.3205291`. (page 30, 48, 84)

[27] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. CuSha: Vertex-Centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14, page 239–252, New York, NY, USA, 2014. Association for Computing Machinery, `https://doi.org/10.1145/2600212.2600227`. (page 10)

[28] Khronos. Open Standard For Parallel Programming of Heterogeneous Systems. https://www.khronos.org/opencl/, 2021. [Online; accessed 10-April-2021]. (page 25)

[29] J. King, T. Gilray, R. M. Kirby, and M. Might. Dynamic Sparse-Matrix Allocation on GPUs. In *International Conference on High Performance Computing*, pages 61–80. Springer, 2016, https://doi.org/10.1007/978-3-319-41321-1_4. (page 10)

[30] H. Liu and H. H. Huang. SIMD-X: Programming and Processing of Graph Algorithms on GPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 411–428, Renton, WA, July 2019. USENIX Association, https://www.usenix.org/conference/atc19/presentation/liu-hang. (page 10)

[31] S. LLC. BlazeGraph. https://www.blazegraph.com/, 2017. [Online; accessed 12-March-2021]. (page 2, 9)

[32] P. E. McKenney and J. D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998. (page 30)

[33] A. McLaughlin and D. A. Bader. Revisiting Edge and Node Parallelism for Dynamic GPU Graph Analytics. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 1396–1406. IEEE, 2014, https://doi.org/10.1109/IPDPSW.2014.157. (page 31)

[34] M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. volume 15, pages 491–504, June 2004, https://doi.org/10.1109/TPDS.2004.8. (page 93)

[35] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM, http://doi.acm.org/10.1145/248052.248106. (page 26, 30)

[36] D. Mlakar, M. Winter, M. Parger, and M. Steinberger. Speculative Parallel Reverse Cuthill-McKee Reordering on Multi- and Many-core Architectures. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IPDPS '21, page 0, 2021, https://doi.org/10.1109/IPDPS49936.2021.00080. (page)

[37] D. Mlakar, M. Winter, P. Stadlbauer, H.-P. Seidel, M. Steinberger, and R. Zayer. Subdivision-Specialized Linear Algebra Kernels for Static and Dynamic Mesh Connectivity on the GPU. In *Computer Graphics forum Volume 39, Issue 2*, EG '20, page 0, 2020, https://doi.org/10.1111/cgf.13934. (page)

[38] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 622–636, New York, NY, USA, 2018. Association for Computing Machinery, https://doi.org/10.1145/3173162.3173180. (page 10)

[39] NVIDIA. nvGraph. https://developer.nvidia.com/nvgraph, 2016. [Online; accessed 12-March-2021]. (page 2, 9)

[40] NVIDIA. NVIDIA CUDA Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2020. [Online; accessed 12-March-2021]. (page 5, 17)

[41] NVIDIA. NVIDIA Volta Architecture Whitepaper. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2020. [Online; accessed 12-March-2021]. (page 5, 95, 118)

[42] M. Parger, M. Winter, D. Mlakar, and M. Steinberger. SpECK: Accelerating GPU Sparse Matrix-Matrix Multiplication through Lightweight Analysis. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 362–375, New York, NY, USA, 2020. Association for Computing Machinery, https://doi.org/10.1145/3332466.3374521. (page)

[43] A. Polak. Counting Triangles in Large Graphs on GPU. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 740–746. IEEE, 2016, https://doi.org/10.1109/IPDPSW.2016.108. (page 31)

[44] A. E. Sariyüce, K. Kaya, E. Saule, and Ü. V. Çatalyürek. Betweenness Centrality on GPUs and Heterogeneous Architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 76–85, 2013, https://doi.org/10.1145/2458523.2458531. (page 31)

[45] T. R. Scogland and W.-c. Feng. Design and Evaluation of Scalable Concurrent Queues for Many-Core Architectures. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 63–74, New York, NY, USA, 2015. ACM, http://doi.acm.org/10.1145/2668930.2688048. (page 30)

[46] M. Sha, Y. Li, B. He, and K.-L. Tan. Accelerating Dynamic Graph Analytics on GPUs. volume 11, page 107–120. VLDB Endowment, September 2017, https://doi.org/10.14778/3151113.3151122. (page 4, 11, 119)

[47] J. Soman, K. Kishore, and P. Narayanan. A Fast GPU algorithm for Graph Connectivity. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010, https://doi.org/10.1109/IPDPSW.2010.5470817. (page 31)

[48] J. Soman and A. Narang. Fast Community Detection Algorithm with GPUs and Multicore Architectures. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 568–579. IEEE, 2011, https://doi.org/10.1109/IPDPS.2011.61. (page 31)

[49] R. Spliet, L. Howes, B. R. Gaster, and A. L. Varbanescu. KMA: A Dynamic Memory Manager for OpenCL. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, pages 9–18, 2014, https://doi.org/10.1145/2588768.2576781. (page 7, 25)

[50] M. Springer and H. Masuhara. DynaSOAr: A Parallel Memory Allocator for Object-Oriented Programming on GPUs with Efficient Memory Access. In A. F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:37, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, https://doi.org/10.4230/LIPIcs.ECOOP.2019.17. (page 7, 27)

[51] M. Steinberger, B. Kainz, B. Kerbl, S. Hauswiesner, M. Kenzel, and D. Schmalstieg. Softshell: Dynamic Scheduling on GPUs. volume 31, pages 161:1–161:11, New York, NY, USA, November 2012. ACM, http://doi.acm.org/10.1145/2366145.2366180. (page 30)

[52] M. Steinberger, M. Kenzel, P. Boechat, B. Kerbl, M. Dokter, and D. Schmalstieg. Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU. volume 33, pages 228:1–228:11, New York, NY, USA, November 2014. ACM, http://doi.acm.org/10.1145/2661229.2661250. (page 30)

[53] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg. ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, San Jose, CA, USA, May 2012. IEEE, https://doi.org/10.1109/InPar.2012.6339604. (page 7, 20)

[54] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. volume 30, pages 202–210, 2005. (page 1)

[55] D. Toedling, M. Winter, and M. Steinberger. Breadth-First Search on Dynamic Graphs using Dynamic Parallelism on the GPU. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2019, https://doi.org/10.1109/HPEC.2019.8916476. (page)

[56] M. Vinkler and V. Havran. Register Efficient Dynamic Memory Allocator for GPUs. volume 34, pages 143–154, 2015, https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12666. (page 7, 23)

[57] H. Wang, L. Geng, R. Lee, K. Hou, Y. Zhang, and X. Zhang. SEP-Graph: Finding Shortest Execution Paths for Graph Processing under a Hybrid Framework on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 38–52, 2019, https://doi.org/10.1145/3293883.3295733. (page 9)

[58] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A High-Performance Graph Processing Library on the GPU. volume 50, page 265–266, New York, NY, USA, January 2015. Association for Computing Machinery, https://doi.org/10.1145/2858788.2688538. (page 2, 9, 15)

[59] S. Widmer, D. Wodniok, N. Weber, and M. Goesele. Fast Dynamic Memory Allocator for Massively Parallel Architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 120–126, 2013, http://doi.acm.org/10.1145/2458523.2458535. (page 7, 21)

[60] M. Winter, R. Zayer, and M. Steinberger. Autonomous, Independent Management of Dynamic Graphs on GPUs. In *2017 IEEE High Performance Extreme Computing Conference (HPEC'17)*, pages 1–7, Waltham, Massachusetts, USA, 2017. University of Technology, Graz, IEEE, https://doi.org/10.1109/HPEC.2017.8091058. (page 48, 119)

[61] M. Winter, D. Mlakar, M. Parger, and M. Steinberger. Ouroboros: Virtualized Queues for Dynamic Memory Management on GPUs. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, New York, NY, USA, 2020. Association for Computing Machinery, https://doi.org/10.1145/3392717.3392742. (page)

[62] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger. FaimGraph: High Performance Management of Fully-Dynamic Graphs under Tight Memory Constraints on the GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, Dallas, Texas, USA, 2018. IEEE Press, https://doi.org/10.1109/SC.2018.00063. (page 6, 7, 84)

[63] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger. Adaptive Sparse Matrix-Matrix Multiplication on the GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 68–81, New York, NY, USA, 2019. Association for Computing Machinery, https://doi.org/10.1145/3293883.3295701. (page)

[64] M. Winter, M. Parger, D. Mlakar, and M. Steinberger. Are Dynamic Memory Managers on GPUs Slow? A Survey and Benchmarks. In *Proceedings of the 26th Symposium on Principles and Practice of Parallel Programming*, PPoPP '21,

pages 68–81, New York, NY, USA, 2021. Association for Computing Machinery, https://doi.org/10.1145/3437801.3441612. (page)