Are Dynamic Memory Managers on GPUs Slow? A Survey and Benchmarks

Martin Winter
University of Technology, Graz
Austria
martin.winter@icg.tugraz.at

Mathias Parger
University of Technology, Graz
Austria
mathias.parger@icg.tugraz.at

Daniel Mlakar
University of Technology, Graz
Austria
daniel.mlakar@icg.tugraz.at

Markus Steinberger
University of Technology, Graz
Austria
steinberger@icg.tugraz.at

Abstract

Dynamic memory management on GPUs is generally understood to be a challenging topic. On current GPUs, hundreds of thousands of threads might concurrently allocate new memory or free previously allocated memory. This leads to problems with thread contention, synchronization overhead and fragmentation. Various approaches have been proposed in the last ten years and we set out to evaluate them on a level playing field on modern hardware to answer the question, if dynamic memory managers are as slow as commonly thought of. In this survey paper, we provide a consistent framework to evaluate all publicly available memory managers in a large set of scenarios. We summarize each approach and thoroughly evaluate allocation performance (thread-based as well as warp-based), and look at performance scaling, fragmentation and real-world performance considering a synthetic workload as well as updating dynamic graphs. We discuss the strengths and weaknesses of each approach and provide guidelines for the respective best usage scenario. We provide a unified interface to integrate any of the tested memory managers into an application and switch between them for benchmarking purposes. Given our results, we can dispel some of the dread associated with dynamic memory managers on the GPU.

PPoPP '21, February 27–March 3, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3441612

CCS Concepts • **Computer systems organization → Single instruction, multiple data**; • **Computing methodologies → Parallel programming languages**; **Massively parallel algorithms**;

Keywords GPU, Memory Management, Survey, Analysis, Benchmarks, CUDA, ScatterAlloc, XMalloc, Halloc, Ouroboros, Bulksemaphore

1 Introduction

One of the major hurdles in converting an existing, dynamic algorithm from the 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 [3, 4, 6] build on one base design, having one arena per CPU core to improve CPU cache hit rates, using a *mutex* to enable concurrent allocation and deallocation and managing memory in chunks. A straightforward port of CPU algorithms, designed to deal with orders of magnitude fewer threads, often does not perform well.

Approaches to manage dynamic memory on GPUs were first proposed around ten years ago with the introduction of dynamic memory management via the NVIDIA Toolkit [13], but there has been renewed interest in the last years due to new hardware capabilities. 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. As modern GPUs can have hundreds of thousands of threads running concurrently, potentially trying to allocate and free memory, many problems surface. These can be thread contention issues, it can include synchronization overhead as well as the ever present problem of fragmentation. Different solutions have been proposed, each focusing on a certain set of these problems, claiming to solve some of the issues. Deciding which approach fits which application best can be a challenge. Furthermore, new

Ref.	Short Name	Year	Availability	Build	Variants	Depend on *CUDA-Alloc.*	General Purpose	Results available	Stable
[9]	*XMalloc*	2010	✗	< 7.0	1	✓	✓	✓	✗
[13]	*CUDA-Allocator*	2010	CUDA API	✓	1	✓	✓	✓	✓
[17]	*ScatterAlloc*	2012	Website	< 7.0	1	✗	✓	✓	✓
[20]	*FDGMalloc*	2013	Website	< 7.0	1	✓	Warp-Level	✗	✗
[19]	*Reg-Eff*	2014	Website	< 7.0	4	✗	✓	✓	✗
[15]	*KMA*	2014	✗	OpenCL	1	✗	✓	✗	?
[1]	*Halloc*	2014	GitHub	< 7.0	1	✓	✓	✓	✓
[16]	*DynaSOAr*	2019	GitHub	✓	1	✗	SOA	✗	?
[7]	*BulkAllocator*	2019	✗	≥ 7.0	2	✗	✓	✗	?
[21]	*Ouroboros*	2020	GitHub	✓	6	✗	✓	✓	✓

Table 1. This table lists all currently available memory managers on the GPU including their build status (does it work natively with independent thread scheduling or even requires it) and how many variants exist. Furthermore, if it depends on the *CUDA-Allocator*, can be used as a general purpose allocator (or has some limitations), if results are available and performance overall was stable throughout testing (if available).

architecture features have been introduced (*e.g.* independent thread scheduling on the NVIDIA Volta [14] architecture). These enable new programming paradigms (*e.g.* blocking algorithms and scheduling guarantees), simplify thread-based computation and ease the conversion of CPU algorithms to the GPU.

Various applications benefit from the use of dynamic memory. This includes dynamic graph analytics (*e.g. cuSTINGER* [8], *aimGraph* [24], *faimGraph* [22] as well as *Hornet* [5] already build on forms of dynamic memory management, either with host intervention or fixed page sizes directly on the GPU), data analytics (*e.g. RAPIDS* [18]), sparse linear algebra (*e.g. AC-SpGEMM* [23]) or databases (*e.g. kinetica* [10]).

In this paper, we attempt to pool the current state-of-the-art in dynamic memory managers on the GPU. This goes back as early as 2010 with *XMalloc* [9], continued with *ScatterAlloc* [17] in 2012, *FDGMalloc* [20] in 2013, an approach by Vinkler and Havran [19] and *Halloc* [1] in 2014 as well as *BulkAllocator* [7] and *Ouroboros* [21] in 2020. 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. Both the evaluation framework as well as all obtained results are available in the GitHub repository. This includes tests of allocation performance, performance scaling, mixed allocation, fragmentation and out-of-memory performance as well as real world testcases, including a synthetic and dynamic graph test case.

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.

2 Approaches

The following section discusses memory managers on the GPU. All of them offer the standard *malloc/free* interface and operate on a block of memory with a configurable size. All follow a similar top-level approach of splitting the available memory into large blocks (mostly fixed size) and use these large blocks to serve the individual allocation requests. Managing these resources varies from the use of lists, queues or even hashing. Any performance evaluation is deferred to Section 4.

2.1 CUDA Allocator

NVIDIA initially introduced its allocator [13] (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 memory aligned to a non-zero power 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 the required allocation size and its thread-based allocation model. It does not natively support any group-based allocation procedures and can only be initialized once with a given size (increasing this memory requires destroying the current context). Reliability is valued over performance.

2.2 XMalloc

XMalloc [9] 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 1. 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 *Memoryblock*. 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 corresponding header information is updated, which might increase internal fragmentation. If a *Basicblock* is completely 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 ScatterAlloc

ScatterAlloc [17] was introduced in 2012 and addresses the problem of collisions during allocation by scattering the allocation requests across its memory regions. 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 whenever 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 speedup 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. The bit-field used is 32 bit long. To support more chunks per page than possible with this field, a second hierarchy level is introduced on the page itself, allowing for a maximum of 1024 chunks per page.

Hashing is used to quickly find new pages and chunks for allocation. This hash function, as can be seen in Figure 2, 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

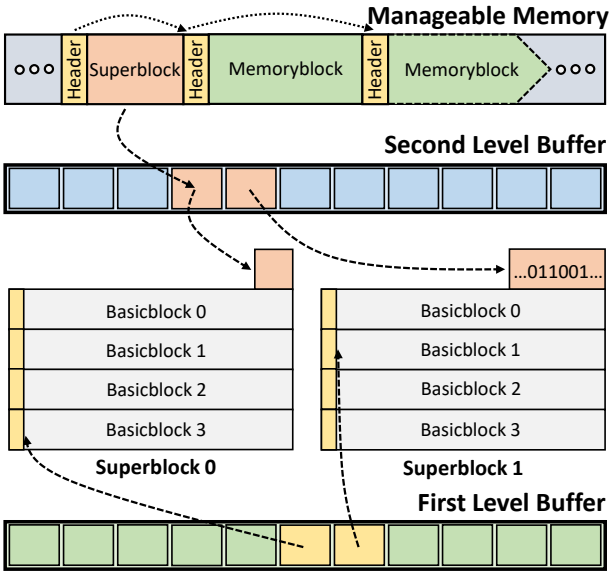


Figure 1. Overview of allocation levels in *XMalloc*.

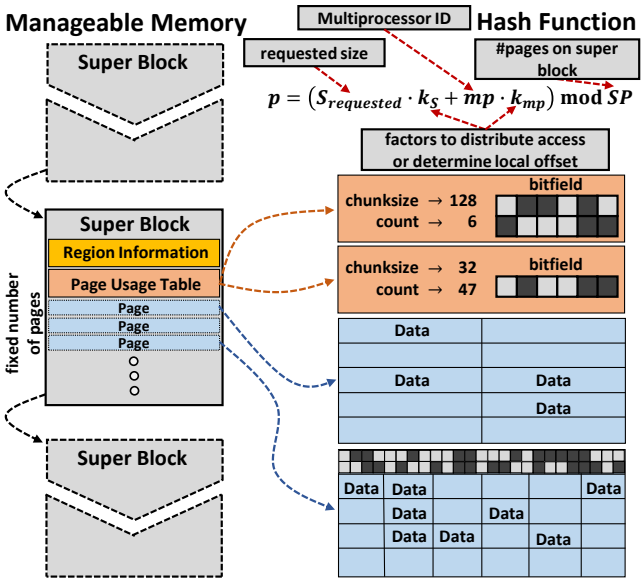


Figure 2. Overview of *ScatterAlloc*.

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 rejected if 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*.

2.4 FDGMalloc

FDGMalloc [20] introduces a memory allocator with a focus on explicit warp-level programming. Their main goal is reducing branch divergence to increase SIMD scalability. They do not offer a general *free* mechanic and only allow allocations at warp-level, reducing its applicability as a general-purpose memory manager. *FDGMalloc* organizes its design in a similar fashion to *ScatterAlloc* and *XMalloc*, by utilizing *SuperBlocks*, which can be split into smaller chunks of memory. The main difference is that within *FDGMalloc*, one *SuperBlock* 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 using the *WarpHeader*, as can be seen in Figure 3. It contains a pointer to the foremost *SuperBlock*, as well as a pointer to a list of *SuperBlocks* that have been allocated using the *CUDA-Allocator*. These lists are of fixed size and are replaced once full. Each list keeps track of how many *SuperBlocks* are already allocated in *SB_Counter* and each *SuperBlock* tracks the number of allocations in *SB_Allocated*. The warp header is allocated from the *CUDA-Allocator* and pointers are distributed to all participating threads. If the total requested size per warp is larger than the maximum *SuperBlock* size, then the request is forwarded to the *CUDA-Allocator*. Otherwise, the current *SuperBlock* is used to allocate the memory. If not all requests can be satisfied within this *SuperBlock*, the remaining threads will once again vote on a leader thread and start allocating a new *SuperBlock*, registering it in the *SuperBlock* list as well.

Deallocation is possible only collectively on a warp-level, there is no way to free single allocations, only all allocations of a warp can be freed simultaneously. Furthermore, to make allocated memory available in successive kernel launches, a pointer to the *WarpHeader* has to be stored in global memory.

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.

2.5 Register Efficient Memory Allocator for GPUs

Vinkler and Havran [19] propose a dynamic memory allocator based on a circular memory pool, organized as a single-linked list. Variants of this will henceforth be called *Reg-Eff*. 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 (an allocation flag and the offset to the next chunk) to enable deallocation. Similar to *ScatterAlloc*, *Reg-Eff* pre-splits the memory into many chunks (except that these chunks need not be uniform in size) to prevent serializing the allocation from a large, initial block at the beginning.

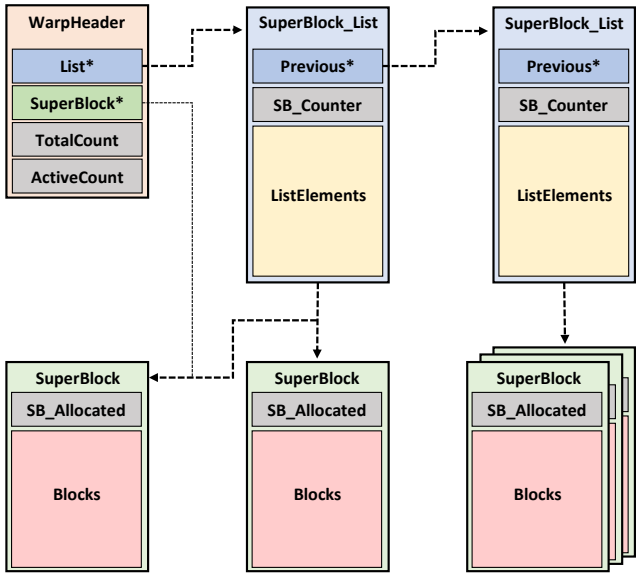


Figure 3. Overview of *FDGMalloc*.

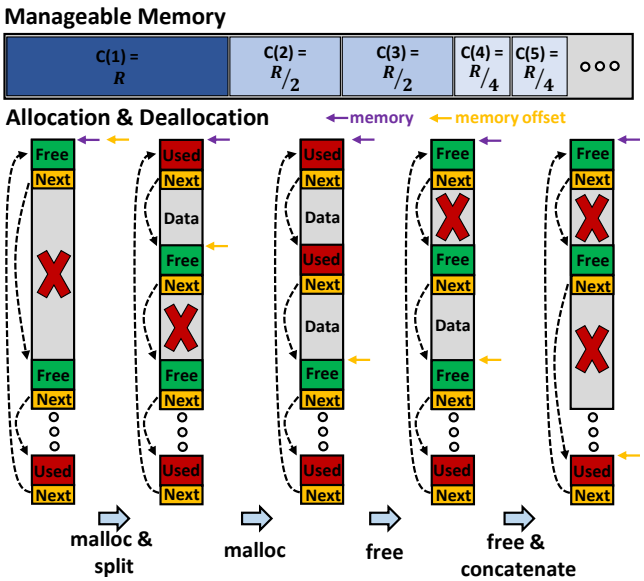


Figure 4. Overview of *Reg-Eff*

This splitting procedure generates a structure similar to a *binary heap*, as can be seen in Figure 4. The memory not used by the heap forms the last chunk.

During allocation, *Reg-Eff* 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 creating the header data for the new chunk, the *memory offset* points 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* (*Reg-Eff - C*), and based on this, three further variants were proposed. *Circular Fused Malloc* (*Reg-Eff - CF*) fuses the two header words into one if less than 2^{31} allocations can be expected. *Circular Multi Malloc* (*Reg-Eff - CM*) and *Circular Fused Multi Malloc* (*Reg-Eff - CFM*) trade fragmentation for speed by introducing an array of offsets (one for each SM) instead of just one shared *memory offset*. This decreases the number of atomic collisions at an increased fragmentation. Additionally, the size of the pre-split chunks is divided by the number of SMs. These smaller heaps are linked in a single-linked list.

2.6 KMA

KMA [15] is built as a two-layer memory manager for OpenCL, with a lower-level generic manager providing direct `malloc` access as well as a high-level manager for managing dynamic data structures. It splits its allocated heap into *Superblocks*, which themselves are split into smaller, power-of-two aligned pages.

During allocation, a fitting *Superblock* is located using a hash map; if none are available, an empty *Superblock* is taken from a free list and initialized. Atomic operations on the *Superblock* state are used to reserve a slot and a free block is located by iterating over the *Superblock* bitmaps. The free list itself is built as a lock-free queue according to Michael et al. [12]. Unfortunately, no source code is available online and the exclusivity to OpenCL exclude this approach from further evaluation.

2.7 Halloc

Halloc [1] starts by allocating *slabs* of 2 MB–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* with one bit for each block that can be allocated from the system.

To allocate a free block, a hash function, as noted in Figure 5, is used to traverse the corresponding bitmap. This visits all blocks and is fast and scalable, as long as ≤85 % of the blocks are allocated. *Warp-aggregated atomics* are used to modify all counters managing the allocation state. 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× 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 can affect performance severely, hence *Halloc* assigns *slabs* to classes. *Free slabs* can switch between chunk sizes, *sparse slabs* (≤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 also 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*.

2.8 DynaSOAr

DynaSOAr [16] deals with the problem inherent with object-oriented programming on the GPU, which is the suboptimal memory layout once objects are layed out in memory as an array of structures (AOS). They propose a fully-parallel, lock-free dynamic memory 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, drastically improving memory access performance, trading memory access speed for allocation speed. As such, it cannot be used as a general-purpose memory manager, as only objects, predefined in their data layout, can be allocated.

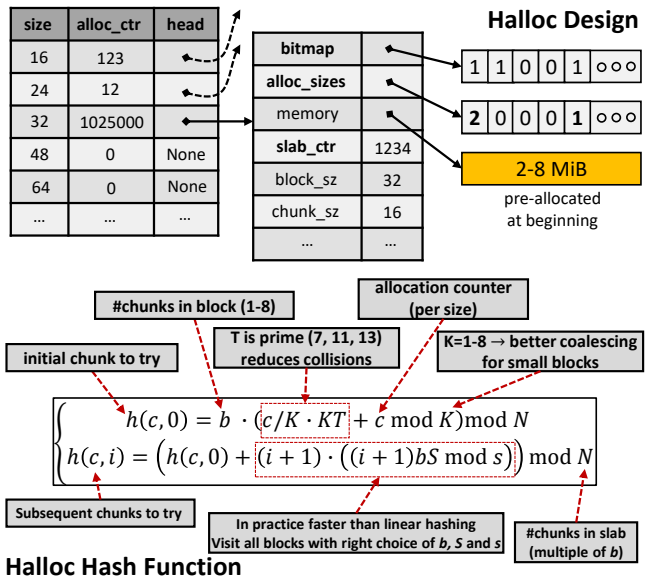


Figure 5. Overview of *Halloc*.

2.9 Throughput-oriented GPU Memory Allocation

BulkAllocator [7] introduces the bulk-semaphore, a throughput-oriented synchronization primitive (*Bulk Semaphore*), as well as two allocators. The bulk semaphore enables preemptive batch allocation, reducing wait times for the allocating threads. 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 modeled 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 list of bins with available elements. The first two bins in a chunk track the allocation state of 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 [11] 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, even after contacting the authors, no public version is available for further testing and replication is challenging due to architecture specific details.

2.10 Ouroboros

Ouroboros [21] extends the queueing concepts and memory manager found in *faimGraph* [22] and instantiates one queue per supported page size. The manageable memory area is split into equally-sized chunks (per default this is 8 KiB), as can be seen in Figure 7. Each queue can either manage pages directly or chunks with free pages. This results in two basic queue types: a *page-based queue* (*Ouro-S-P*) and a *chunk-based queue* (*Ouro-S-C*). The *page-based queue* is fast and efficient, but lacks the reusability of chunks once they have been assigned to a page size. The *chunk-based queue* trades allocation speed for memory efficiency. It has a two-stage access design (allocate from chunk in queue) but can efficiently reuse empty chunks for all purposes. One drawback of these two queues is their memory requirements, as both need static space, which has to be large enough to hold the largest expected number of free pages/chunks.

To reduce the static memory requirements, *Ouroboros* virtualizes the queues by shifting the queue storage from a static region onto dynamic chunks. Two variants are introduced, one using a small chunk pointer array to reference the chunks currently allocated to the virtual queue (called *virtualized array hierarchy queue*, shorthand *Ouro-VA-P* and *Ouro-VA-C*). The other version gets rid of this chunk pointer array altogether in favor of pointers to the beginning and end of the virtual queue (called *virtualized linked-chunk queue*, shorthand *Ouro-VL-P* and *Ouro-VL-C*). Multiple instances of *Ouroboros* (with different page size ranges) can be instantiated simultaneously to allow for larger allocation sizes, otherwise larger allocations are relayed to the *CUDA-Allocator*.

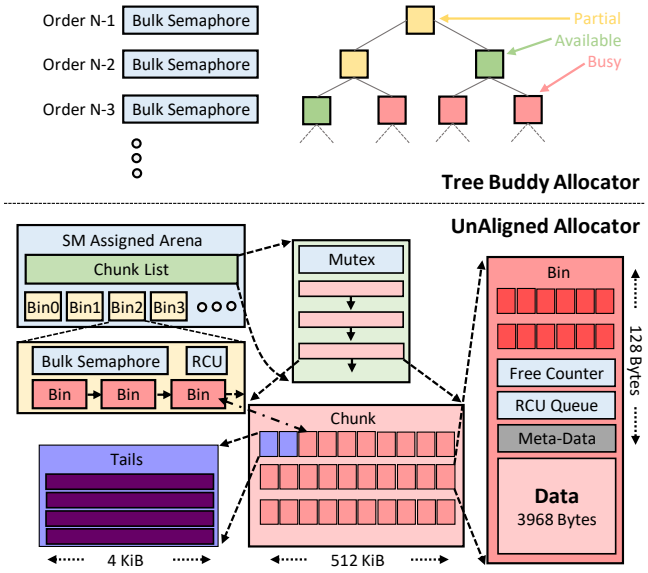


Figure 6. Overview of the *BulkAllocator*.

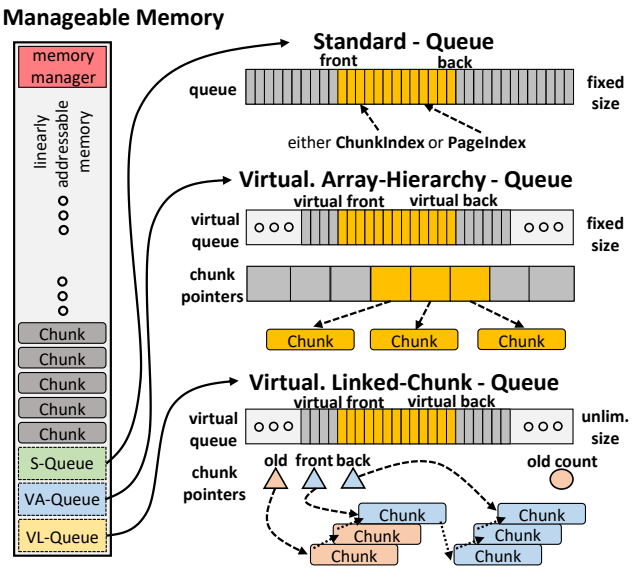


Figure 7. Overview of *Ouroboros*.

3 Framework

As part of this survey, we provide our complete test framework as well as an interface to all tested memory managers. This includes *CUDA-Allocator*, *XMalloc*, *ScatterAlloc*, *Halloc*, *Reg-Eff* and *Ouroboros*. *FDGMalloc* is also included, but crashes in most test scenarios, 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. 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 testsuite (including results for the NVIDIA TITAN V and NVIDIA RTX 2080Ti) can be found on GitHub.

4 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 8.2.1 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 [14]. For these, we pass *compute_60* to the compiler to enforce warp-synchronous execution.

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 *Reg-Eff* 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. *Reg-Eff* includes a simple memory manager (referred to as *Atomic*) just building on atomics on a shared offset, this is used as a baseline when applicable. We use a consistent color scheme throughout all plots to save on space, this color map can be seen in Figure 8. Due to page limitations, we only show a subset of all captured results in the paper, all plots (except for Figure 9e and Figure 9f, which were captured on the NVIDIA RTX 2080Ti) show evaluation results from the NVIDIA TITAN V. The full set of plots can be found in the Appendix and on GitHub.

4.1 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* (~6 ms). All other approaches are close in initialization performance (30 ms–40 ms), except for *Halloc*, which is on about 5.5× slower compared to the average initialization time.

We also evaluate register requirements for *malloc* and *free* respectively. The respective *malloc* implementation requires more registers than *free* for all approaches. The four variants of *Reg-Eff*, 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 very large discrepancy between *malloc* (168) and *free* (24).

4.2 Allocation Performance

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

- Allocation performance (thread/warp-based)
- Allocation performance for mixed sizes (thread-based)
- Performance scaling for varying numbers of threads for powers of two between 2^0 - 2^{20}

4.2.1 Allocation Performance for Allocation Size

We test 10.000 and 100.000 allocations in the range between 4 B–8192 B. Figure 9 shows the resulting performance plots for thread-based allocations as well as for warp-based allocations (one thread per warp allocates). Results obtained on the NVIDIA RTX 2080Ti follow the overall trend as evaluated on the NVIDIA TITAN V, as can be seen when comparing Figure 9c & Figure 9e and Figure 9d & Figure 9f. Henceforth we will showcase results only from the TITAN V for sake of brevity (full results can be found on GitHub).

The performance results suggest that the *CUDA-Allocator* also 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. 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

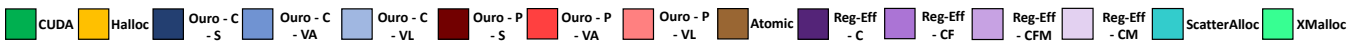
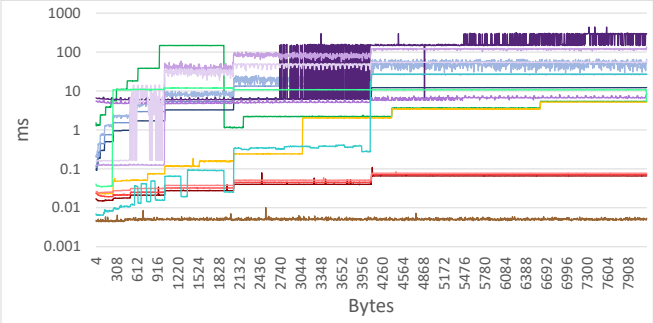
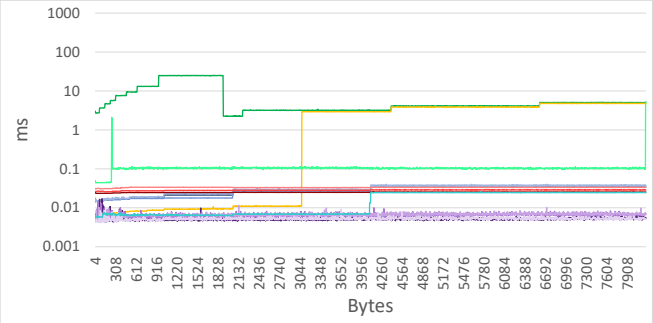


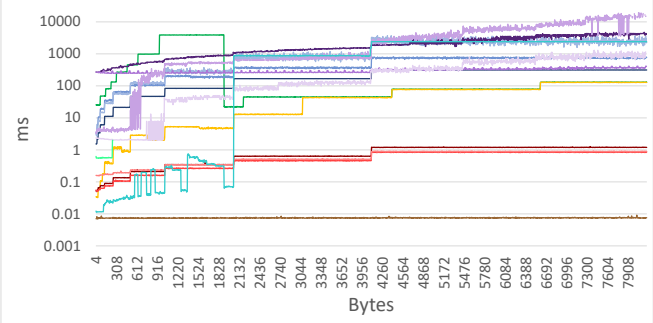
Figure 8. Color scheme used henceforth for all tested approaches.



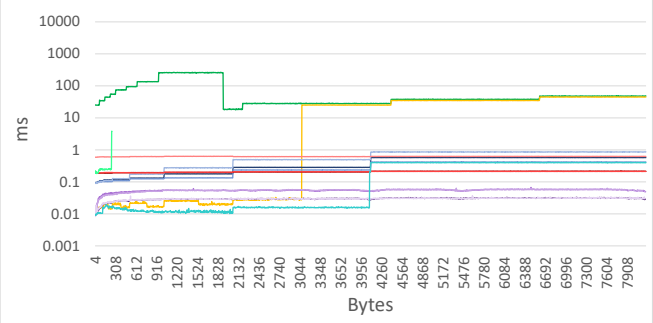
(a) [Titan V] Thread-based allocation performance 10K



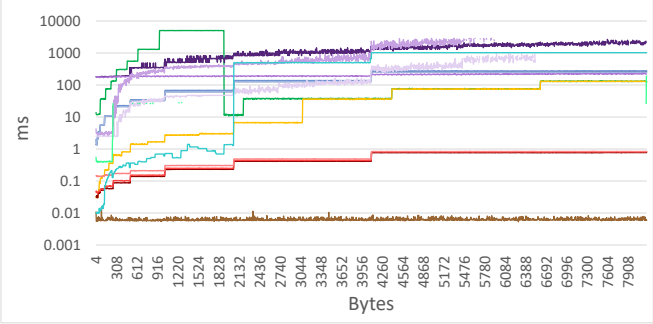
(b) [Titan V] Thread-based deallocation performance 10K



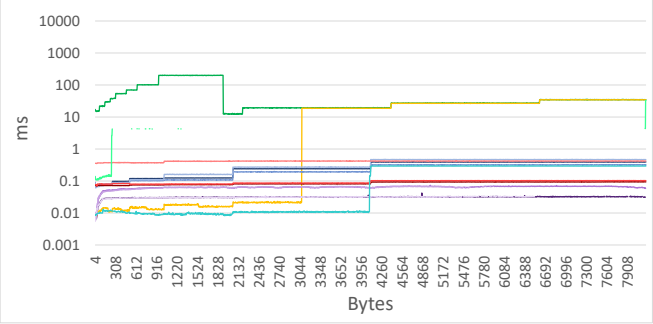
(c) [Titan V] Thread-based allocation performance 100K



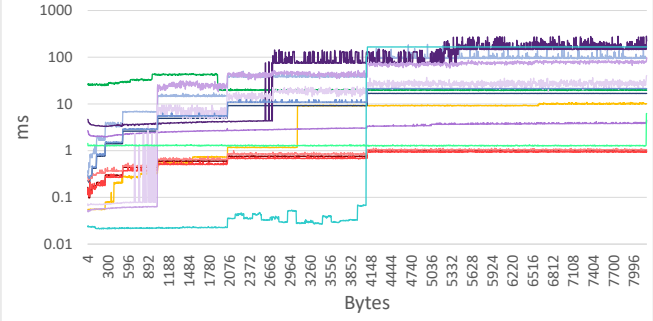
(d) [Titan V] Thread-based deallocation performance 100K



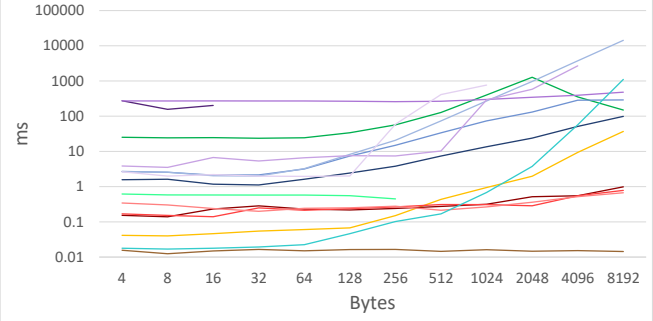
(e) [2080Ti] Thread-based allocation performance 100K



(f) [2080Ti] Thread-based deallocation performance 100K



(g) Warp-based allocation performance 10K



(h) Mixed allocation performance 100K

Figure 9. Thread-based allocation/deallocation performance for 10 000 (9a and 9b on the NVIDIA TITAN V) as well as for 100.000 (**9c** and **9d** on the NVIDIA TITAN V and **9e** and **9f** on the NVIDIA RTX 2080Ti) allocations for the range 4 B–8192 B, as well as warp-based allocation performance for 10.000 (**9g**) allocations and mixed allocation for 100.000 (**9h**) allocations in the range 4 B to 4 B–8192 B (4 B–4 B, 4 B–8 B, ⋯ , 4 B–8192 B).

performance consistently above 1 ms. *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 considerable 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*. *Reg-Eff* 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 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.

Warp-based allocation changes the picture somewhat, as can be seen in Figure 9g, in that *Ouroboros* slows down a little, while *Reg-Eff* 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-Reg-Eff* 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*.

4.2.2 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 B–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 9h.

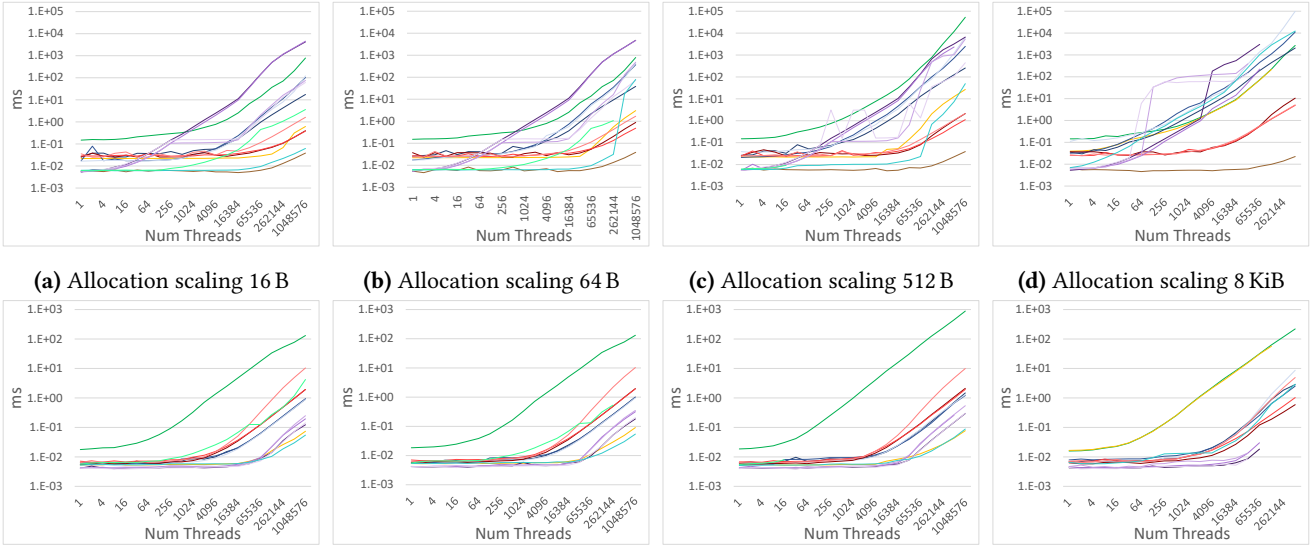
Considering smaller allocation ranges, *ScatterAlloc* clearly performs best once again, followed by *Halloc* and *page-based Ouroboros*. After increasing the range to 4 B–1024 B, *page-based Ouroboros* clearly shows its strength. The *CUDA-Allocator* shows its characteristic spike at 2048 B, after which performance increases again.

4.2.3 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 10a to Figure 10d.

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. *Halloc*, *ScatterAlloc* as well as *page-based Ouroboros* remain flat for one order of magnitude longer, but especially increasing allocation sizes decrease performance for both *Halloc* and *ScatterAlloc*, while *page-based Ouroboros* shows a consistent profile over the full range.

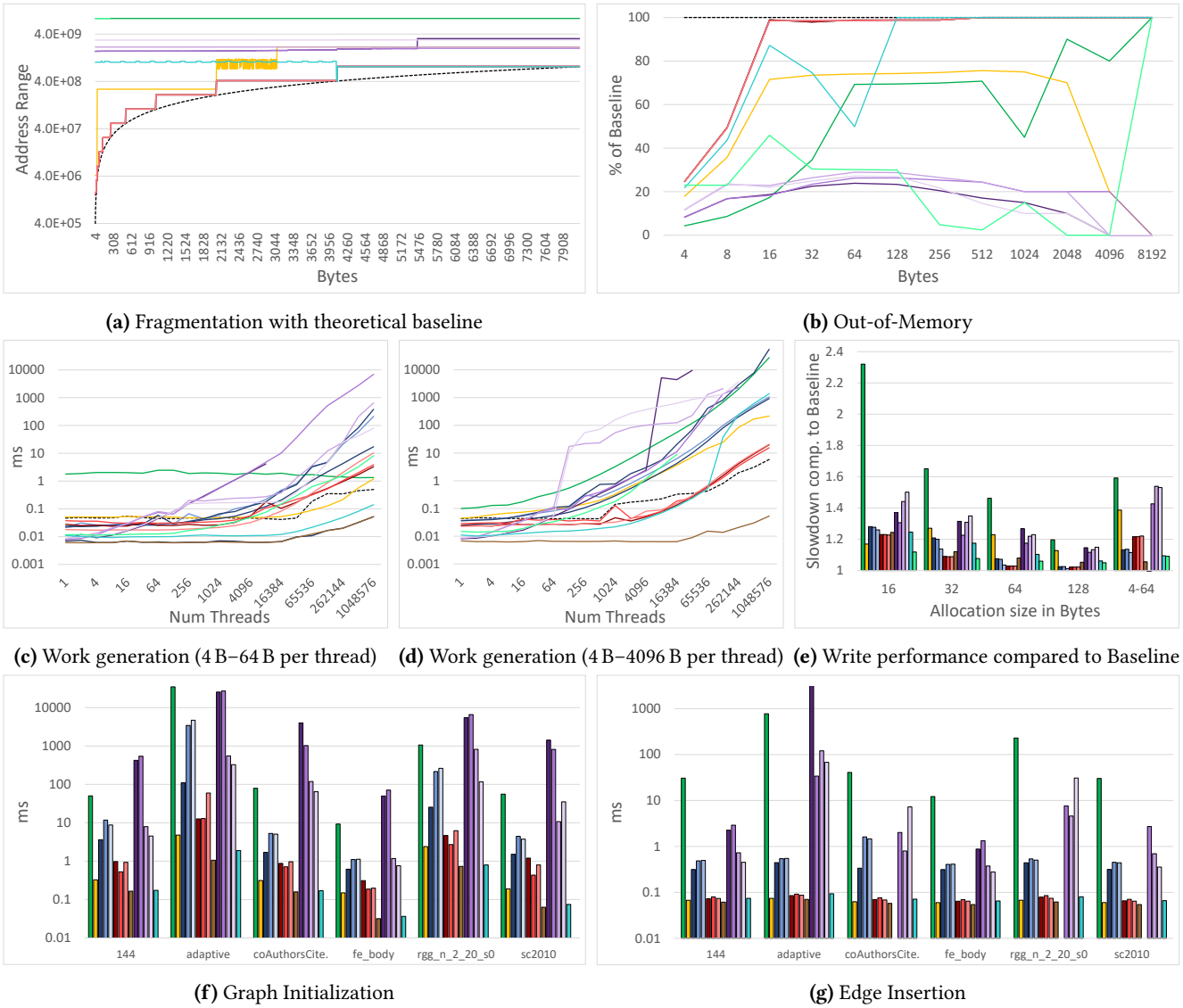
Reg-Eff shows an unusual pattern as it starts decreasing in performance much earlier compared to the other approaches,



(a) Allocation scaling 16 B **(b)** Allocation scaling 64 B **(c)** Allocation scaling 512 B **(d)** Allocation scaling 8 KiB

(e) Deallocation scaling 16 B **(f)** Deallocation scaling 64 B **(g)** Deallocation scaling 512 B **(h)** Deallocation scaling 8 KiB

Figure 10. Allocation performance scaling for 16 B, 64 B, 512 B and 8 KiB (**10a** - **10d**), as well as deallocation performance (**10e** - **10h**).



(a) Fragmentation with theoretical baseline

(b) Out-of-Memory

(c) Work generation (4 B–64 B per thread)

(d) Work generation (4 B–4096 B per thread)

(e) Write performance compared to Baseline

(f) Graph Initialization

(g) Edge Insertion

Figure 11. Fragmentation testcase reporting the address range returned after 100.000 allocations (**11a**), out-of-memory testing for various sizes (**11b**), two synthetic workload testcases with 4 B–64 B (**11c**) and 4 B–4096 B (**11d**) per thread (*Baseline* uses *Thrust prefix-sum*). We also compare write performance to allocated memory to the *Baseline* for the range 16 B–128 B (**11e**) and initializing a graph (**11f**) and inserting 100.000 edges (**11g**) focused on a small range of source vertices.

even for small thread counts, as can be seen in Figure 10a and Figure 10b. Figure 10d shows the performance discrepancy between *page-based Ouroboros* and all other variants very clearly for larger allocation sizes. Considering deallocation performance in Figure 10e to Figure 10h, 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.

4.3 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.

4.3.1 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 11a. *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 *Reg-Eff*.

4.3.2 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 11b 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* 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 *Reg-Eff* do not finish but are reigned 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.

4.4 Real-World Performance

We evaluate three real-world test scenarios: work generation, graph initialization as well as updating a dynamic graph.

4.4.1 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 different ranges, 4 B–64 B (as can be seen in Figure 11c) of work generated per thread as well as 4 B–4096 B (in Figure 11d). We launch an increasing number of threads and also compare to the *Baseline* built on a prefix-sum from *Thrust*. For the smaller range, as in Figure 11c, 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 11d, *Halloc* slows down, with only *ScatterAlloc* and *page-based Ouroboros* outperforming the *Baseline* up to tens of thousands of threads.

4.4.2 Memory Access Performance

On the GPU, not only allocation speed but also memory access speed is crucial. To evaluate whether a memory allocator considers 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 11e, *Ouroboros* stays closest to the fully coalesced baseline, closely followed by *XMalloc*, *ScatterAlloc* and *Halloc*. *Reg-Eff* and the *CUDA-Allocator* show poor access times.

4.4.3 Graph Initialization

We test graph initialization performance for a set of graphs taken from the *DIMACS10 graph data set* [2]. Each adjacency is aligned to a power of two and the results can be seen in Figure 11f. *CUDA-Allocator* performs worst in all scenarios, followed by the variants of *Reg-Eff* and *chunk-based Ouroboros*. *Halloc* and *page-based Ouroboros* perform similarly, once again beaten by *ScatterAlloc*, as most graphs are sparse and require many small allocations.

4.4.4 Graph Updates

We also consider updating the graph. As soon as an existing adjacency crosses over a power of two barrier during the allocation change, we allocate a new adjacency and free the old adjacency. We test two different scenarios, uniform updates as well as updates focused on a range of source vertices, to simulate more update pressure, which can be seen in Figure 11g. This testcase also highlights the ability for concurrent allocations and deallocations. Once again, the *CUDA-Allocator* takes the most amount of time, followed by *Reg-Eff* and then *chunk-based Ouroboros*. *Page-based Ouroboros*, *Halloc* and *ScatterAlloc* all perform equally well in this case.

5 Discussion

Based on our detailed evaluation in Section 4, we 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 4.4.1. 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.

Reg-Eff 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 ≤ 2048 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.

6 Conclusion

In this paper, we surveyed currently available dynamic memory managers on the GPU. We provide a test framework, which includes all evaluated memory managers with a consistent interface for straightforward integration into existing projects. Our evaluation leads to the following conclusions:

- Thread-based Allocation
  - If an application mainly requires small allocations (≤ 512 B), *ScatterAlloc* is the clear choice with *Halloc* and the *page-based Ouroboros* staying close.
  - Larger allocations (≤ 2048 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 B
  - *Halloc* also improves its performance, outperforming page-based *Ouroboros* up to 1024 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 *Reg-Eff* 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*, *Reg-Eff* 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.

Acknowledgments

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

References

[1] Andrew V Adinetz and Dirk Pleiter. 2014. Halloc: a high-throughput dynamic memory allocator for GPGPU architectures. In *GPU Technology Conference (GTC)*, Vol. 152.

[2] D. A. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner. 2014. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*. AMS, Atlanta, Georgia, USA, 73–82.

[3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. 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)*. Association for Computing Machinery, New York, NY, USA, 117âĂŞ128. https://doi.org/10.1145/378993.379232

[4] Jeff Bonwick and Jonathan Adams. 2001. Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*. USENIX Association, USA, 15–33.

[5] F. Busato, O. Green, N. Bombieri, and D. A. Bader. 2018. Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, Waltham, Massachusetts, USA, 1–7.

[6] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, Ottawa, Canada*.

[7] Isaac Gelado and Michael Garland. 2019. Throughput-oriented GPU Memory Allocation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 27–37. https://doi.org/10.1145/3293883.3295727

[8] O. Green and D. Bader. 2016. cuSTINGER: Supporting dynamic graph algorithms for GPUs. In *2016 IEEE High Performance Extreme Computing Conference (HPEC '16)*. Georgia Institute of Technology, IEEE, Waltham, Massachusetts, USA.

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

[10] Kinetica DB Inc. 2020. *kinetica Active Analytics Platform: GPU-Accelerated Database*. https://rapids.ai [Online; accessed 10-August-2020].

[11] Paul E McKenney and John D Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, Vol. 509518.

[12] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*. Association for Computing Machinery, New York, NY, USA, 267–275. https://doi.org/10.1145/248052.248106

[13] NVIDIA. 2020. NVIDIA CUDA Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/. (2020). [Online; accessed 08-July-2020].

[14] NVIDIA. 2020. NVIDIA Volta Architecture Whitepaper. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. (2020). [Online; accessed 10-August-2020].

[15] Roy Spliet, Lee Howes, Benedict R. Gaster, and Ana Lucia Varbanescu. 2014. KMA: A Dynamic Memory Manager for OpenCL. In *Proceedings of Workshop on General Purpose Processing Using GPUs (GPGPU-7)*. Association for Computing Machinery, New York, NY, USA, 9–18. https://doi.org/10.1145/2588768.2576781

[16] Matthias Springer and Hidehiko Masuhara. 2019. DynaSOAr: A Parallel Memory Allocator for Object-Oriented Programming on GPUs with Efficient Memory Access. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, Alastair F. Donaldson (Ed.), Vol. 134. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 17:1–17:37. https://doi.org/10.4230/LIPIcs.ECOOP.2019.17

[17] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. 2012. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *Innovative Parallel Computing (InPar), 2012*. IEEE, San Jose, CA, USA, 1–10. https://doi.org/10.1109/InPar.2012.6339604

[18] RAPIDS Development Team. 2018. *RAPIDS: Collection of Libraries for End to End GPU Data Science*. https://rapids.ai

[19] M. Vinkler and V. Havran. 2015. Register Efficient Dynamic Memory Allocator for GPUs. *Computer Graphics Forum* 34, 8 (2015), 143–154. https://doi.org/10.1111/cgf.12666 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12666

[20] Sven Widmer, Dominik Wodniok, Nicolas Weber, and Michael Goesele. 2013. Fast Dynamic Memory Allocator for Massively Parallel Architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU-6)*. 120–126. https://doi.org/10.1145/2458523.2458535

[21] Martin Winter, Daniel Mlakar, Mathias Parger, and Markus Steinberger. 2020. Ouroboros: Virtualized Queues for Dynamic Memory Management on GPUs. In *Proceedings of the 34th ACM International Conference on Supercomputing (ICS '20)*. Association for Computing Machinery, New York, NY, USA, Article 38, 12 pages. https://doi.org/10.1145/3392717.3392742

[22] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2018. 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)*. IEEE Press, Dallas, Texas, USA, Article Article 60.

[23] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2019. Adaptive Sparse Matrix-Matrix Multiplication on the GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 68–81. https://doi.org/10.1145/3293883.3295701

[24] M. Winter, R. Zayer, and M. Steinberger. 2017. Autonomous, independent management of dynamic graphs on GPUs. In *2017 IEEE High Performance Extreme Computing Conference (HPEC'17)*. University of Technology, Graz, IEEE, Waltham, Massachusetts, USA, 1–7.

A Artifact Description Appendix

A.1 Abstract

The following appendix provides all necessary information to download the framework used to evaluate all memory managers, rerun the experiments as well as access the full results acquired on the NVIDIA TITAN V as well as from the NVIDIA RTX 2080Ti.

A.2 Description

A.2.1 How the software can be obtained

The framework is downloadable from Zenodo and exists also as a GitHub repository.

A.2.2 Hardware dependencies

The framework was tested on an Intel Core i7-7700 as well as an Intel Core i7-8700X, with 32 GB RAM respectively. We also tested different cards from NVIDIA, including the GTX 1080Ti, the TITAN V as well as the RTX 2080Ti (the repository includes results for the TITAN V as well as the 2080Ti).

A.2.3 Software dependencies

The framework was tested on Windows 10, Arch Linux <5.9.9> as well as Manjaro <5.4>.

- **C++ compiler**
  - Tested with **gcc-9**, **gcc-10** as well as **msvc 16.8.X**
- CUDA Toolkit
  - Tested with **10.1**, **10.2**, **11.0** and **11.1**
- CMake
  - Tested with ≥**3.16**
- Python
  - Tested with ≥**3.8**
  - Requires package **argparse**
- Boost
  - Tested with **1.66** and **1.74**
- Git

A.2.4 Datasets

Graphs for the real world testcases can be downloaded from the SuiteSparse Matrix Collection.

A.3 Installation

To install the framework, first make sure all the requirements are installed and configured correctly (this includes setting the path to the **boost** install location on Windows in `BaseCmake.cmake`).

- **Setup**
  - **Option A**
    - Download archive from Zenodo, extract it and call the following commands in the top-level directory:
      - `git submodule init`
      - `git submodule update`
  - **Option B**
    - Clone respository from GitHub, pass the option `--recursive` and choose branch `AEsubmission`
- Call `python init.py`
- **Install**
  - Use the `Developer PowerShell` on Windows
  - **Option A**:
    - To build all tests at once, call
      - `python setupAll.py --cc XX`
      - cc → Compute capability (tested `61`, `70` and `75`)
  - **Option B**:
    - To build each testcase individually, locate `setup.py` in each testfolder and call
      - `python setup.py --cc XX`
      - cc → Compute capability (tested `61`, `70` and `75`)

A.4 Experiment workflow

Since running the whole testsuite takes quite a long time (≥ 600h) for a full run, the framework also includes a script to run a smaller, representative testsuite. To run this, simply call

- `python testAll.py -mem_size X -device Y -runtest -genres`
  - The memory size is given in GB
  - The device ID is an integer (0 is the default device)

To run individual commands, please take a look at the `README.md` file found in the top-level directory, which describes each testcase as well as all parameters that can be changed and which format is expected. Example commands are listed in Table 2.

A.5 Evaluation and expected result

All commands required to reproduce all results featured in this paper can be found in Table 2. Running these commands will generate `.csv` files similar to those found in `results/TITANV` or `results/2080Ti`. Generating the plots unfortunately is not automatized, each of those results folders also holds all `.xlsx` files used to generate the plots found in the paper. One can copy over newly generated results into the corresponding `Excel` file to also generate the plots. The `README.md` file as well as Table 2 also reference which script produces results for which plot found in the paper.

A.6 Experiment customization

Each of the listed testcases (see Table 2) can be modified in many different ways, an exact listing with all options can be found in the `README.md` file. All testcases have a few options in common:

- `-t o+s+c+h+r+x`
  - One can select per testcase which framework should perform the testcase, each is selected by the first letter of the approach and multiple approaches are chained using the `+` symbol.

Figure/Section	Folder	Script	Command
-	All folders	Common to all All files end in `.py`	`-t o+s+h+c+r+x -device 0 -allocsize 8 -runtest -genres -timeout 120`
Section 4.1	`synth_tests`	`test_registers`	`python test_registers.py <...>`
Section 4.1	`synth_tests`	`test_synth_init`	`python test_synth_init.py <...>`
Figure 9c & Figure 9d	`alloc_tests`	`test_allocation`	`python test_allocation.py -num 100000 -range 4-8192 -iter 100 -timeout 120 <...>`
Figure 9g	`alloc_tests`	`test_allocation`	`python test_allocation.py -num 10000 -range 4-8192 -warp -iter 100 -timeout 120 <...>`
Figure 9h	`alloc_tests`	`test_mixed_allocation`	`python test_mixed_allocation.py -num 100000 -range 4-8192 -iter 100 -timeout 120 <...>`
Figure 10a - Figure 10h	`alloc_tests`	`test_scaling`	`python test_scaling.py -threadrange 0-20 -byterange 4-8192 -iter 100 -timeout 300 <...>`
Figure 11a	`frag_tests`	`test_fragmentation`	`python test_fragmentation.py -num 100000 -range 4-8192 -iter 100 -timeout 60 <...>`
Figure 11b	`frag_tests`	`test_oom`	`python test_oom.py -num 100000 -range 4-8192 -timeout 3600` **`-allocsize 2`** `<...>`
Figure 11c	`synth_tests`	`test_synth_workload`	`python test_synth_workload.py -range 4-64 -threadrange 0-20 -iter 100 -timeout 300 <...>`
Figure 11d	`synth_tests`	`test_synth_workload`	`python test_synth_workload.py -range 4-4096 -threadrange 0-20 -iter 100 -timeout 300 <...>`
Figure 11e	`synth_tests`	`test_synth_workload`	`python test_synth_workload.py -range 4-4096 -threadrange 0-20 -iter 100 -timeout 300 -testwrite <...>`
Figure 11f	`graph_tests`	`test_graph_init`	`python test_graph_init.py -timeout 600 -configfile config_init.json`
Figure 11g	`graph_tests`	`test_graph_update`	`python test_graph_update.py -timeout 600 -configfile config_update_range.json`

Table 2. All testcases have a few parameters in common (**which are ommited in the table to save space and denoted by `<...>` for each script**). This includes the device selection (`-device 0`) and the managable memory size for each memory manager (`allocsize 8`, given in GB, all started with 8 GB, except for out-of-memory testcase, which is started with 2 GB). For each test script one can also select which approaches to test, given by the initial letter of the approach, chained together using the + symbol (to run all, pass `-t o+s+h+c+r+x`). To run a test, generating new results, pass the option `-runtest` and to aggregate all individual results into one file, pass the option `-genres`. Both can be combined, which will run the tests first and the aggregate the results into one file. One can specify a timeout for each individual run of a testcase (pass `-timeout 60`, given in seconds), after which the process will be killed by the OS. Details for each testcase can be found in `README.md`.

- `-runtest`
  - Runs the testcase for the selected approaches and generates new results
- `-genres`
  - Collects produced results and aggregates them in one file
- `-timeout 60`
  - Kills an individual testcase after so many seconds
- `-device 0`
  - Selects which device to use
- `-allocsize 8`
  - Specifies the size of the manageable memory per approach in GB