Ouroboros: Virtualized Queues for Dynamic Memory Management on GPUs

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

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

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

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

ABSTRACT
Dynamic memory allocation on a single instruction, multiple threads architecture, like the Graphics Processing Unit (GPU), is challenging and implementation guidelines caution against it. Data structures must rise to the challenge of thousands of concurrently active threads trying to allocate memory. Efficient queueing structures have been used in the past to allow for simple allocation and reuse of memory directly on the GPU but do not scale well to different allocation sizes, as each requires its own queue.

In this work, we propose Ouroboros, a virtualized queueing structure, managing dynamically allocatable data chunks, whilst being built on top of these same chunks. Data chunks are interpreted on-the-fly either as building blocks for the virtualized queues or as paged user data. Re-usable user memory is managed in one of two ways, either as individual pages or as chunks containing pages. The queueing structures grow and shrink dynamically, only currently needed queue chunks are held in memory and freed up queue chunks can be reused within the system. Thus, we retain the performance benefits of an efficient, static queue design while keeping the memory requirements low. Performance evaluation on an NVIDIA TITAN V with the native device memory allocator in CUDA 10.1 shows speed-ups between 11× and 412×, with an average of 118×. For real-world testing, we integrate our allocator into staymGraph, a dynamic graph framework with proprietary memory management. Throughout all memory-intensive operations, such as graph initialization and edge updates, our allocator shows similar to improved performance. Additionally, we show improved algorithmic performance on PageRank and Static Triangle Counting.

Overall, our memory allocator can be efficiently initialized, allows for high-throughput allocation and offers, with its per-thread allocation model, a drop-in replacement for comparable dynamic memory allocators.

1 INTRODUCTION
Many algorithms are challenging to port to the Graphics Processing Unit (GPU) due to their dynamic nature. Various approaches try to deal with unpredictable, dynamic execution requirements and varying parallelism. Inherent to those—but also present in many applications with fixed parallelism—are dynamic memory requirements. 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 architectures, like the GPU, thousands of concurrently active threads may access and reallocate resources.

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 GPU for memory allocation. The combination of thousands of active GPU threads with the requirement to keep memory fragmentation low and avoid CPU round-trips poses a significant challenge.

Various approaches have been proposed to keep track of a large number of dynamic objects concurrently. 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. As
any number of objects can be linked, linked-lists are only limited by the available memory. However, especially on the GPU, linked-lists come with many disadvantages. 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.

Array-based methods, on the other hand, are—due to their static size—inherently limited by the number of objects they can manage. 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, often with a large variance or even unpredictable number of objects, which requires significant overallocation of the arrays. Even if the maximum required size for a specific input can be precomputed, it may be reached for just a fraction of the applications runtime, leading to overall underutilization of the available memory.

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. Chunks are broken up into pages, managed by one array-based queue per page size. To reduce the memory overhead, we virtualize the array-based 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 [22]. 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. Bulk allocations of pages are handled using an optimized synchronization primitive [5].
- We propose two virtualized queues, storing the array-based queues themselves on memory chunks. Both can either manage page indices directly for maximum allocation 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.
- We integrate Ouroboros into famiGraph [22] to create Ouro-Graph, demonstrating the applicability and benefits of our allocator in a real application scenario.

Compared to the library-provided CUDA Allocator, Ouroboros achieves speed-ups between 11× and 412×. Evaluating fragmentation, Ouroboros achieves more efficient utilization compared to Halloc [1], ScatterAlloc [19] as well as the CUDA Allocator.

Compared to famiGraph [22], we reduce memory requirements on average by 23× and show improved initialization and algorithmic performance while retaining comparable update performance.

2 RELATED WORK

Most related to our efforts are other dynamic memory allocators and queues designed for the GPU.

2.1 Allocators

NVIDIA included dynamic memory allocation on the GPU with their Fermi architecture in 2009 [14], OpenCL [8] does not provide an allocator in the language. However, the CUDA allocator is often regarded as slow and unreliable [10, 19], indicating that dynamic allocation is difficult on the GPU. The first published dynamic allocators for the GPU were XMalloc [10] and ScatterAlloc [19]. The former builds on lock-free FIFO queues that hold chunks and bins of predefined sizes, which can be subdivided further to fulfill smaller requests. Operations over memory blocks require locks. The latter is built over a fixed size ring buffer of pages and uses bitmaps to perform allocations on those pages. Hashing is used to scatter requests to different pages to reduce congestion.

Following these early allocators, FDGMalloc [21] is similar to XMalloc, but combines allocations within a warp in a non-standard interface. Halloc [1] subdivides its fixed-size memory pool into chunks during initialization and stores them in per-bin-size lists. Memory management essentially moves data between lists, large allocations are relayed to the CUDA allocator. Finally, aiming for low register usage, simple allocation strategies that may lead to higher fragmentations can be used [20]. Only recently, dynamic GPU memory management has again received attention, as novel GPU hardware guarantees support for blocking algorithms. This enables the use of bulk semaphores to reduce interfering concurrent operation during allocation [5].

While the variety among allocation strategies is high, all allocators rely on either lists or queues to manage resources. In general, those are used per chunk size and thus instantiated many times.

2.2 Queues

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 [6] in 1983. Parallel GPU designs have long focused on non-blocking linked-lists, including the famous Michael-Scott queue [13] and the Shann-Huang-Chen queue [4]. With larger parallelism, the ordering between elements enqueued concurrently is practically irrelevant, which can be exploited by putting elements into the same bucket [9].

Specialized GPU queue designs are scarce. Task-based GPU runtime systems have used proprietary solutions, mixing linked-lists and array-based queues [17, 18]. Scogland and Feng proposed a blocking array-based GPU queue [15]. Unfortunately, their queue blocks on empty states making it impracticable for memory allocation. The BrokerQueue [11] removes these blocking states by pairing concurrent enqueues and dequeues. Blocking only happens to ensure ordering between pairs. While link-based queues also work on the GPU, their performance is multiple orders of magnitude lower than array-based designs [11].
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, we combine the best of both strategies. It is as efficient as an array-based queue, but is completely built on dynamic memory.

3 BUILDING BLOCKS AND BACKGROUND
In the following, we discuss the building blocks essential to Ouroboros and their relevance to the system.

Figure 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.

3.1 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, memory must be used efficiently. (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 or can be provided by the user. Should the given allocation be too small, the system has the option to automatically re-initialize in a larger area of memory. All allocation requests are handled directly on the GPU, avoiding costly round-trips to the CPU. A memory manager keeps track of all available resources and offers standard malloc and free functionality for individual threads.

The designated dynamic memory is subdivided into equally-sized chunks, which can be allocated from the memory manager in O(1). The size of these chunks can be matched to the specific application (standard size is 8 KiB). 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 CUDA Allocator, can be used or Ouroboros’s chunks could be integrated into the page-table system (by a vendor).

3.2 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 metadata (padded to multiples of 128 B) and a large region to hold data (both specific to the actual use case). Chunks are used in three different ways, as shown in Figure 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. A chunk can be allocated to a different page size or different chunk type if completely freed.
- **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.

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 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.
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.

3.3 Queues

We utilize an array-based method for memory reclamation in form of an index queue. Queues are not only used for memory reuse, but also to reuse other dynamic objects, which includes QueueChunks. Algorithm 1 demonstrates the basic enqueue and dequeue functionality of the queues.

Both first test the fill_count to check the viability of the operation (enqueue 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 (atomicCAS) 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.

Algorithm 1: 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()

3.4 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. As access primitive we use a bulk semaphore [5]. 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. 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 2:

- 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 2 are implemented using an atomicCAS operation, since different 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 new value.
4 QUEUES FOR MEMORY MANAGEMENT

The most basic tool for memory management are efficient index queues \cite{11, 22}. These are usually implemented as array-based queues, operating on top of a ring buffer. 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. 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, 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 or chunks (containing available pages). One instance manages one or the other, but Ouroboros can combine multiple instances with different items.

4.1 Queues managing pages

This queue type is the most straightforward evolution of the index queue, as it also stores page indices directly. The fill counter is replaced by a bulk semaphore to allow for efficient 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. The bulk semaphore allows for efficient and interleaved allocation of new pages from chunks. Algorithm 3 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 signaled 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 the page index read, as detailed in Section 3.3. Deallocations work exactly as dequeue() described in Algorithm 1, replacing the fill count with a bulk semaphore.

This design excels in terms of allocation speed, but bears some disadvantages. 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. Consequently, the allocation from the chunk pool takes more time, as all pages are added to the queue by one thread.

4.2 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

Algorithm 3: Allocate / Free page with the page-based queue

\begin{verbatim}
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)
\end{verbatim}

Algorithm 4: Allocate / Free page with the chunk-based queue

\begin{verbatim}
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, 1, #pages
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)
\end{verbatim}
Algorithm 5: Allocate page from chunk

```
Function Chunk::allocatePage(index)
  while curr_count ← atomicSub(count, 1) ≤ 0 do
    if curr_count ← atomicAdd(count, 1) < 0 then
      return CONTINUE_TRAVERSAL
    else if curr_count = 0 then
      mode ← RE_ENQUEUE
    if curr_count = 1 then
      if mode = RE_ENQUEUE then
        mode ← SUCCESS
      else
        mode ← DEQUEUE
    else if mode ≠ RE_ENQUEUE then
      mode ← SUCCESS
    mIndex ← 0
  while True do
    mask = bitmask[mIndex]
    while lowestbitset ← findFirstSet(mask) do
      bits ← createPattern(lowestbitset)
      mask ← atomicAnd(mask[mIndex, bits])
      if checkBitSet(mask, lowestbitset) then
        return mode
    mIndex ← (mIndex + 1) mod maskSize
```

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, both the fill counter and the bulk semaphore are used. 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 4, 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. 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 5. In case of failure, the current thread advances in the queue to the next chunk and repeats the query until successful. Once all pages on a chunk have been allocated, it can be removed from the queue (line 18 in Algorithm 4). 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 4) 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 5. 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 4). The last deallocation on a chunk (line 30 in Algorithm 4) 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 marker, the queue location is simply skipped during allocation. The clear focus of this queue type is memory efficiency, as it requires less queue storage on average. Comparing performance to the page-based variant, the two-stage approach will typically perform worse, as the current front will not be advanced as fast (line 19 in Algorithm 4) given a high number of concurrent threads, leading to queue traversal.

4.3 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. 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. 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 VIRTUALIZED QUEUES FOR MEMORY MANAGEMENT

The aforementioned queues potentially suffer from large 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. 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.

5.1 Virtualized Array-Hierarchy Queue (VAQ)

A VAQ replaces statically allocated queues by a much smaller chunk pointer queue (Figure 2, middle). 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. Each thread still determines queue positions using atomics on the front and back pointer. However, these positions are 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.

The thread assigned to position 0 on a QueueChunk preemptively 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 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. 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. Once the correct QueueChunk has been located, the low-level enqueue and dequeue operations follow the same principle as detailed in Algorithm 1.

Each QueueChunk has two counters to determine the current fill-level (countA and countB in Figure 2), 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. Such a QueueChunk is returned to 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.

### 5.2 Virtualized Linked-Chunk Queue (VLQ)

The VLQ (Figure 2, right) replaces the 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 size limit, as queues can grow arbitrarily large and shrink to virtually nothing. 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. The thread with position 0 on a QueueChunk again preemptively allocates the next chunk. 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.

### Algorithm 6: Move front/back pointer along

```plaintext
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
```

If countA = #spots after an enqueue, all enqueues on this chunk have been finished and the back pointer can be moved using Algorithm 6. Due to the potentially high 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 atomicCAS of another has failed).

A dequeue operation starts from front. 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 6. Additionally, we count the successful moves.

Moving the front pointer does not remove the corresponding QueueChunks. This is crucial, as threads may still be reading from this QueueChunk during traversal. While we could use hazard pointers [12], 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 ≥ t chunks behind front. t is determined heuristically from the number of potentially concurrently-active threads, which is limited for a given GPU.
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. 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 10.1.243.

We first evaluate allocation performance, comparing Ouroboros to the native device allocation functionality provided by CUDA, as well as Scatteralloc and Halloc (Section 6.1). Then, we present results of a real-world example (Section 6.2), integrating Ouroboros and CUDA Allocator into fainGraph [22], a dynamic graph framework, which performs allocation of fixed-size pages on the GPU. Ouroboros is initialized with a queue capacity of 2,000,000 elements and a chunk size of 8 KiB. This results in 10 different queues for allocations in the range 16 B–8192 B. Ouroboros follows the standard memory manager interface using malloc/free to allocate or free memory on the device, as can be seen in Algorithm 7.

6.1 Evaluation of allocation performance

To investigate allocation performance, we compare against the device allocation functionality provided by NVIDIA and also briefly highlight Scatteralloc [19] as well as Halloc [1]. For all test scenarios, the frameworks were instantiated with 2 GiB of manageable memory.

6.1.1 Allocation performance. We test two allocation counts (10,000 and 100,000 allocations) and allocation sizes from 16 B–8192 B and a mixed case with allocations from within that range. Performance results averaged over 100 runs are given in Figure 3.
Interestingly, performance of the native CUDA allocator can be divided into three intervals: (1) allocations ≤ 64B, the time per allocation is constant, indicating a padding to 64B; (2) from 64B–1024B, the allocation time increases with size; (3) a 2048B allocation is approximately 110–160× faster than a 1024B allocation, followed again by an increase in time. Unfortunately, NVIDIA does not provide any information about the internals of their allocator. Thus, one can only speculate about the cause of this behavior. 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.

Contrary, Ouroboros profits from memory reuse and shows performance increases when allocating from a (partially) filled queue. Comparing our page-based approaches to the CUDA Allocator, performance is on average 165× better for 10 000 allocations and 274× better for 100 000 allocations. Comparing our chunk-based approaches to the CUDA Allocator, performance is on average 15× better for 10 000 allocations and 18× better for 100 000 allocations. Our chunk-based methods perform better up to the split at 1024B and only fall behind for larger sizes. Our page-based methods always significantly outperform CUDA Allocator. Note that our virtualized methods show little variance compared to their fixed-size array-based counterparts, indicating the effectiveness of our virtualization strategies.

We also tried to evaluate Halloc and ScatterAlloc, but both experience issues on newer hardware. To gather results we explicitly enforced warp-synchronous behavior for both. Furthermore, Halloc fails on some allocation sizes tested (and even for those that work, allocating different sizes during one kernel call also fails). In light of these caveats, a comparison to Halloc results in a performance ratio \( t_{\text{halloc}} / t_{\text{ouroboros}}\) for the completed cases, which is on average 1.68× (page-based) and 0.15× (chunk-based) for 10 000 allocations and 1.74× (page-based) and 0.16× (chunk-based) for 100 000 allocations.

Compared to ScatterAlloc, the performance ratio is on average 4.33× (page-based) and 0.30× (chunk-based) for 10 000 allocations and 12.5× (page-based) and 1.12× (chunk-based) for 100 000 allocations. These good results for Ouroboros are surprising, as its focus is on reducing the memory footprint and fragmentation, while those allocators trade fragmentation for allocation speed.

6.1.2 Performance scaling. To evaluate performance scaling over the number of calling threads, we test four representative allocation sizes (32B, 128B, 1024B and 8192B), as can be seen in Figure 3. The CUDA Allocator performs consistently worse than all other competitors up to 1024B, overall showing a consistent decrease in allocation performance over the number of threads with all tested allocation sizes. After 1024B, as already detailed in Section 6.1.1, the performance resets to a similar level as with 32B. Halloc performs well for 32B, but already at 128B we see the larger thread counts fail, which is exacerbated at 1024B. At 8192B it internally relays allocation calls to the CUDA Allocator.

ScatterAlloc performs well for the first two testcase, but starts to slow down for the larger allocation sizes and becomes even slower than the CUDA Allocator for the largest tested size.

Our chunk-based methods perform better than the CUDA Allocator up until 1024B, where they then fall in line with ScatterAlloc. Our page-based methods perform well for all four tested allocation sizes, being on-par with Halloc and ScatterAlloc for the first two sizes while gaining the performance edge for 1024B and greatly outperforming all other for 8192B.

6.1.3 Fragmentation. To test memory efficiency, the first testcase continuously allocates 16B elements until out-of-memory. Ouroboros manages to utilize 98.35% of the given memory. ScatterAlloc comes second with 87.1%, followed by Halloc with 84.5% and the CUDA Allocator with just 17.2%—indicating that CUDA Allocator has some internal limitations on either the number of allocations or allocations of small size.

Figure 3 shows two further fragmentation tests. The testcase on the bottom left evaluates the maximum address range returned by the allocator for 100 000 allocations of the given size. The CUDA Allocator reveals an interesting pattern; the returned address are always nearly exactly the allocated amount of memory apart, which is especially important if one wants to use parts of the entire memory for other data, e.g., to store dynamic vertices in a dynamic graph framework.

The testcase on the bottom right performs concurrent allocations/deallocations of 16B (freeing just 50% of the allocations each iteration, resulting in a 50% growth), tracking the maximum address range until out-of-memory. This once again highlights the efficiency of Ouroboros, outlasting ScatterAlloc by about 200 iterations and Halloc by about 300 iterations. Interestingly, the CUDA Allocator slows down with each iteration, about 7s before failure. ScatterAlloc maintains performance except for the last iterations, while Ouroboros and Halloc maintain performance throughout.

6.2 Evaluation of dynamic graph scenario

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 CudaAllocGraph. Using these allocators, adjacency data is stored on contiguous memory pages, compared to the linked-list of pages used in

---

**Algorithm 7: End-to-end usage example**

```c
// Configure an instance
using MemoryManager = Ouroboros<OuroborosPages-Params..., OuroborosChunks-Params..., ...>

// GPU code
1 Function __global__ deviceKernel(MemoryManager mm)
2 auto ptr = mm.malloc(size);
3 // Memory stays persistent over kernel launches
4 mm.free(ptr);

// CPU code
5 MemoryManager mm(SIZE);
6 deviceKernel<<<>>>(mm);
```
were only modified to account for different adjacency traversals. We can also compare to other graph framework, like OuroGraph.

Table 1: Graph data set used for dynamic graph performance evaluation, taken from the 10th DIMACS Graph Implementation Challenge [2].

<table>
<thead>
<tr>
<th>Name</th>
<th>Vertices</th>
<th>Edges</th>
<th>Adj. Mean</th>
<th>Adj. Std. Dev.</th>
<th>Adj. Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>luxembourg_osm</td>
<td>114599</td>
<td>239332</td>
<td>2.08</td>
<td>0.41</td>
<td>6</td>
</tr>
<tr>
<td>coAuthorsCiteseer</td>
<td>227320</td>
<td>1628268</td>
<td>7.16</td>
<td>10.63</td>
<td>1372</td>
</tr>
<tr>
<td>coAuthorsDBLP</td>
<td>229067</td>
<td>1953552</td>
<td>6.54</td>
<td>9.82</td>
<td>336</td>
</tr>
<tr>
<td>Idoor</td>
<td>952203</td>
<td>46522475</td>
<td>48.86</td>
<td>11.95</td>
<td>77</td>
</tr>
<tr>
<td>audiwk_1</td>
<td>943695</td>
<td>77651847</td>
<td>82.28</td>
<td>42.45</td>
<td>345</td>
</tr>
<tr>
<td>delaunay_n20</td>
<td>1048576</td>
<td>6291372</td>
<td>6.0</td>
<td>1.34</td>
<td>23</td>
</tr>
<tr>
<td>rgg_n_2_20_s0</td>
<td>1048576</td>
<td>13783240</td>
<td>13.14</td>
<td>3.63</td>
<td>36</td>
</tr>
<tr>
<td>hugetric-00000</td>
<td>5824554</td>
<td>17467046</td>
<td>3.0</td>
<td>0.03</td>
<td>3</td>
</tr>
<tr>
<td>delaunay_n23</td>
<td>838608</td>
<td>50331568</td>
<td>6.0</td>
<td>1.34</td>
<td>28</td>
</tr>
<tr>
<td>germany_osm</td>
<td>11548845</td>
<td>24738362</td>
<td>2.14</td>
<td>0.53</td>
<td>13</td>
</tr>
<tr>
<td>nlpkt120</td>
<td>3542400</td>
<td>46845792</td>
<td>27.34</td>
<td>3.09</td>
<td>28</td>
</tr>
<tr>
<td>nlpkt200</td>
<td>16240000</td>
<td>448225632</td>
<td>27.6</td>
<td>2.42</td>
<td>28</td>
</tr>
<tr>
<td>nlpkt240</td>
<td>27993600</td>
<td>774472352</td>
<td>27.66</td>
<td>2.22</td>
<td>28</td>
</tr>
<tr>
<td>europe_osm</td>
<td>50912018</td>
<td>108109320</td>
<td>2.12</td>
<td>0.48</td>
<td>13</td>
</tr>
</tbody>
</table>

faimGraph. This speeds up the adjacency access and manipulation significantly and further simplifies the framework interface. The rest of the faimGraph framework remains unchanged. We omitted other frameworks due to prior mentioned issues. With this setup, we can also compare to other graph framework, like aimGraph [23], cuSTINGER [7], Hornet [3] and GPMA [16]. 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. The used graph data set is listed in Table 1.

6.2.1 Initialization. faimGraph and OuroGraph are initialized similarly: both first determine memory requirements in parallel and then efficiently write adjacencies. For CudaAllocGraph, 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 OuroGraph and faimGraph is that OuroGraph flushes the complete memory with deletion markers first, such that all chunks can also be used as QueueChunks without prior initialization. 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 4.

Both, faimGraph and OuroGraph, outperform CudaAllocGraph 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 flush 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 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.

6.2.2 Edge Updates. 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, denoted random and pressure in Figure 4 respectively. 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). CudaAllocGraph 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 overallocation on the existing allocations. Nevertheless, OuroGraph offers simplified and more efficient edge update procedures.

The comparison in Figure 4 shows the clear advantage in allocation performance of OuroGraph over CudaAllocGraph. 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. 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. CudaAllocGraph performs worst in all cases, being orders of magnitude slower for larger graphs.

6.2.3 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. 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 without the need to convert the adjacency traversal to a proprietary structure.

PageRank. Performance comparison using PageRank (Figure 4) 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 balance 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 4. On average, performance is 80% higher (ranging from 3%–200%) comparing OuroGraph to faimGraph. When starting whole warps per adjacency, we can more efficiently read in adjacency data and use

---

**Figure 4:** Dynamic graph performance for OuroGraph variants (S|VA|VL denote standard, virtualized array-based and virtualized linked-list based methods, P|C define if page or chunk indices were stored), faimGraph and CudaAllocGraph. Top Left: Initialization performance for given graphs in ms, Top Right: Memory footprint after initialization in MB, Middle: Two test-cases, updating 100,000 edges with randomized source (second from the top) and 100,000 edges with source focused on a range of 1000 (third from the top), insertion performance on the right and deletion performance on the left. Bottom Left: Algorithm performance for PageRank and Bottom Right: Algorithm performance for STC.
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. $CudaAllocGraph$ falls short on update performance in all cases. $OuroborosGraph$ is close to $faimGraph$’s update performance, but significantly reduces memory requirements and boosts algorithmic performance.

7 CONCLUSION & FUTURE WORK
Dynamic memory allocation on GPUs is a long standing and much researched topic. $Ouroboros$ 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. Allocation performance compared to the native CUDA allocator shows a speed-up of 118x on average. Incorporating $Ouroboros$ into $faimGraph$ shows improved initialization times, memory footprint and algorithmic run-times on $PageRank$ and $STC$ and shows promising results for edge updates as well.

In the future, we consider investigating queue compaction techniques, such that our page-based queues can profit from chunk reuse, independent of the prior chunk type. Additionally, we will add group-based allocation, such that groups of threads (warps, blocks or arbitrary size) can allocate pages efficiently together without the need for separate allocations.

Overall, our evaluation suggests that $Ouroboros$ not only shows great performance over the full allocation range, but can also, due to its per-thread allocation model, serve as a drop-in replacement for currently used device memory allocators. By making $Ouroboros$ and its future derivatives open source, we hope to inspire further work on dynamic memory management on GPUs and thereby shape the future of parallel algorithms.

ACKNOWLEDGMENT
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