Throughput-Oriented GPU Memory Allocation

Isaac Gelado
NVIDIA
Santa Clara, California
igelado@nvidia.com

Michael Garland
NVIDIA
Santa Clara, California
mgarland@nvidia.com

Abstract

Throughput-oriented architectures, such as GPUs, can sustain three orders of magnitude more concurrent threads than multicore architectures. This level of concurrency pushes typical synchronization primitives (e.g., mutexes) over their scalability limits, creating significant performance bottlenecks in modules, such as memory allocators, that use them. In this paper, we develop concurrent programming techniques and synchronization primitives, in support of a dynamic memory allocator, that are efficient for use with very high levels of concurrency.

We formulate resource allocation as a two-stage process, that decouples accounting for the number of available resources from the tracking of the available resources themselves. To facilitate the accounting stage, we introduce a novel bulk semaphore abstraction that extends traditional semaphore semantics by optimizing for the case where threads operate on the semaphore simultaneously. We also similarly design new collective synchronization primitives that enable groups of cooperating threads to enter critical sections together. Finally, we show that delegation of deferred reclamation to threads already blocked greatly improves efficiency.

Using all these techniques, our throughput-oriented memory allocator delivers both high allocation rates and low memory fragmentation on modern GPUs. Our experiments demonstrate that it achieves allocation rates that are on average 16.56 times higher than the counterpart implementation in the CUDA 9 toolkit.

Keywords Concurrency, Memory Allocation, GPU Programming

1 Introduction

Modern throughput-oriented architectures, exemplified by NVIDIA GPUs, are used to accelerate applications in a wide range of domains, including computer graphics, scientific computation, and machine learning. Such architectures rely on massive multithreading as the key to efficient execution; for example, over the last decade, the maximum number of simultaneous threads has risen from 6,144 on the NVIDIA G80 to 172,032 on the NVIDIA GV100 architectures, respectively. Utilizing such large thread populations is relatively straightforward when applications exhibit abundant data parallelism. However, applications that rely on concurrent data structures, and accompanying synchronization primitives (e.g., mutexes) that arbitrate access to them, present a greater challenge. Because many techniques for managing concurrent data access have been designed to suit the needs of multiprocessor systems with 2–3 orders of magnitude fewer threads, we find they often perform poorly when faced with the level of contention that can arise on GPUs.

In this paper, we explore the design of a throughput-oriented memory allocator suitable for implementing the standard C `malloc` and `free` interfaces available to threads running on the GPU. It represents both a critical building block for applications requiring dynamic memory management and an exemplar of the class of techniques that require concurrent access to shared data structures. Because there can be many tens of thousands of threads running concurrently, we focus on maximizing the total (de)allocation *rate*, measured by (de)allocation calls completed per second. We simultaneously seek to minimize memory fragmentation, which can grow rapidly at high allocation rates if left unchecked.

Currently few GPU applications use dynamic memory, however a high performance allocation will benefit GPU software in domains such as graph analytics (e.g., Gunrock [23]), data analytics (e.g., RAPIDS [2]), sparse linear algebra (e.g., cuSolver [8]), or databases (e.g., kinetica [1]). Often time programmers allocate an upper bound array in the host to circumvent the low performance device allocator. This results in a waste of memory and limits the application dataset sizes. To fit larger datasets, programmers rely on a two-phase approach: a first stage computes the amount of memory required, and a second phase performs the actual computation. A high throughput device allocator removes the refactoring of algorithms, and would remove the need for pre-computations.

Contention within the allocator can be extremely high, since any one of the many concurrent threads on the GPU—up to 172,032 on current architectures—may call `malloc` and `free` at any time. Therefore, our allocator design is focused

PPoPP '19, February 16–20, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6225-2/19/02…$15.00
https://doi.org/10.1145/3293883.3295727

on techniques for contention management, all of which we believe are also applicable to the design of most concurrent data structures on the GPU. First, we design shared data structures that allow **fine-grained mutual exclusion** regions. We track free memory at coarse granularity using a buddy system [14], based on a static binary tree, that requires locking *at most* two nodes to perform any update. By minimizing locking, many coarse-grained operations can proceed concurrently. Second, we exploit **parallel execution within critical sections**. Concurrent threads in a given kernel may often allocate memory blocks of the same size. Rather than serializing their execution, we group multiple threads to execute a parallel lookup to find multiple free blocks within the same critical section using *collective synchronization primitives*. For example, a cooperating group of threads can collectively acquire a mutex together, work collectively to perform an operation, and collectively release the mutex when all are finished. This both reduces contention and exploits local parallelism within critical sections. Third, we **delegate execution** to already waiting threads rather than forcing many threads to wait as well. For example, we extend our Read-Copy-Update [18] (RCU) mechanism for managing linked lists of memory blocks to permit threads to delegate commit operations to any other thread already waiting to commit.

Applying these principles, we have designed a new high-throughput memory allocator for massively multithreaded architectures such as the GPU. Our CUDA C++ implementation, benchmarked on an NVIDIA Titan V, delivers substantially higher allocation throughput compared to the `malloc` and `free` implementations provided in the CUDA 9 Toolkit; in our benchmarks delivering an allocation rate between 0.22 and 346 times faster, with and average 16.56X improvement.

The first contribution of this paper are a novel synchronization primitive, *bulk semaphores*, that generalizes counting semaphores and enables efficient concurrent allocation of resources. The second contribution of this paper is a extension of RCU to enable delegation of clean-tasks that avoids unnecessarily blocking the execution of threads. The third contribution are *collective synchronization primitives*, the first synchronization construct that enables parallel execution within critical sections. Finally, the forth contribution of this paper is a new high-throughput GPU memory allocator that relies on our other contributions to achieve high scalability.

2 Background

Dynamically allocating memory is a collaboration between the operating system (OS) and a user-level memory allocator. The OS allocates regions of the virtual address space of the application and commits physical pages as needed to provide storage resources. The user-level memory allocator—invoked by the application through interfaces such as `malloc`—subdivides regions provided by the OS to fulfill application requests.

Most modern user-level memory allocators [4, 6, 11] share the same high-level architecture. *Arenas* manage memory in large fixed-size *chunks*; for example, *jemalloc* [11] uses 2 MB chunks. The arena also defines a set *bins*, each of which corresponds to a fixed allocation size, and includes metadata to locate available blocks of that size. To support concurrent (de)allocation, the arena is protected by a synchronization primitive (e.g., a mutex) that is locked prior to starting any operation and released upon completion.

The number of arenas is typically chosen to be a multiple of the number of CPU cores [11] or the expected number of threads [4]. A thread is assigned an arena upon its first call to the allocator, and all subsequent allocations are served from this arena. This association of arenas to cores/threads enables concurrent allocations and improves CPU cache hit rates [4], albeit at the cost of higher memory fragmentation.

2.1 GPU Execution Model

Our goal is to develop an allocator design suitable for throughput-oriented processors that rely on massive multi-threading to achieve high performance. We focus specifically on NVIDIA's Volta architecture [19] because it is the first generation of GPU architecture to support *independent thread scheduling*, which guarantees forward progress for individual threads regardless of control flow. This guarantee greatly simplifies the implementation of thread-level synchronization primitives.

We use CUDA C++ for our prototype implementation. CUDA applications execute tasks on the GPU in the form of *grids*, which organize the potentially many threads of a kernel into *thread blocks*. All threads within a single thread block share access to on-chip memory and can efficiently synchronize with each other using hardware-supported barriers. Each thread block is placed on a given Streaming Multiprocessor (SM), the hardware unit on which all threads of that block will run.

Host code running on the CPU can reserve portions of GPU memory using the `cudaMalloc` interface, which fills the role of the OS-level allocator described above and which we use unmodified. Individual threads running on the GPU request dynamic allocation by calling `malloc`, and it is through this interface that our implementation is exposed to the application.

2.2 Scalable Memory Allocation

The first GPU memory allocator, XMalloc [12], is based on lock-free FIFO queues that hold both available chunks and bins of pre-defined sizes. Chunks are allocated from blocks of contiguous memory that can be sub-divided into arbitrary sizes. XMalloc keeps a list of available memory blocks separated by boundary tags, and operations over memory blocks require locking. We use the same allocation strategy as XMalloc; a coarse-grained allocator makes large allocations, and fulfills requests from a fine-grained allocator that handles small allocations.

ScatterAlloc [20] also implements a similar architecture but relies on the CUDA dynamic memory allocator to allocate chunks and handle large allocations. ScatterAlloc tracks memory availability using bitmaps. To prevent collisions when atomically updating a bitmap, ScatterAlloc defines a hashing function that scatters the atomic operations over the words of the bitmap. We use a similar technique to scatter the traversal of a static binary tree in our coarse grained allocator to prevent collisions when locating memory blocks.

Widmeret al. [24] built a memory allocator similar to XMalloc, but they defined a non-standard per-warp allocation interface to coalesce (de)allocation requests. We implement the standard `malloc` and `free` interface, and we transparently coalesce requests within the allocator by detecting which threads are concurrently invoking it and using specialized paths for single-threaded and full-warp operations.

Vinkler et al. [22] studied different dynamic memory allocation algorithms focusing on minimizing the number of registers used by the allocator code and proposed a simple allocator based on incrementing a free pointer in the memory pool to be used as a coarse-grained allocator. This approach lead to large memory fragmentation. We use a buddy system as the coarse-grained allocator, which is known to keep fragmentation low [14]. We track free memory using a binary tree, similarly to Marotta et al. [16], but we couple *bulk semaphores* to this data structure to throttle down the number of concurrent updates to the tree.

HAlloc [3] is the fastest GPU memory allocator currently available. It defines a statically sized memory pool that gets sub-divided into chunks at initialization time, and relies on the CUDA dynamic memory allocator to handle large allocations. For each allocation size, HAlloc keeps only one active bin from which to allocate. Whenever usage within a bin reaches a configurable threshold, HAlloc replaces it with a new active bin. This strategy maximizes the chances of subsequent allocations finding an available block in the active bin. HAlloc also keeps per-size lists of bins that are almost-exhausted, and of bins that are almost-empty. Free operations move bins between this two lists, and bins in the almost-empty list are used to select new active bins when needed. Our fine-grained allocator also keeps per-size bins, but we use a linked-list to track all active bins, and thus avoid costly active bin replacement operations.

3 Concurrent Resource Management

In this section we describe the synchronization primitives we use in our GPU memory allocator, semaphores and RCU, and discuss how we can adapt them to accommodate the massive amounts of concurrency and the thread execution model of modern GPUs. Section 4 explains how we use these primitives to aid in the construction of our memory allocator.

3.1 False Resource Starvation

Resource allocators often group available resources into pools (e.g., chunks in *jemalloc*). One could conceive a pool design using a lock-free or a wait-free data structure, e.g., a wait-free queue [15]. Initially, the queue contains a single memory block that encompasses all the available memory. A first thread dequeues this memory block to allocate the requested memory from it. If another thread allocates memory, it would find the pool empty, and return an *out-of-memory* error. If this thread would have waited for the first one to enqueue the modified block, the allocation would have succeeded. We refer to this problem as *false resource starvation*.

The simplest scheme to prevent a resource manger from running into the *false resource starvation* problem is to serialize (de)allocation operations. Most allocators implement these operations within a critical section protected by a mutex. However, serialization results in unacceptable performance when the number of concurrent operations is high. Most concurrent CPU memory allocators rely on serving only a small number of threads (or even one) from each memory arena. This design is not suitable for a GPU since the total number of threads is far too large. For example, using 32 threads per arena and a chunk size of 512KB would require preallocating up to 5GB of memory for the arenas.

We develop **two-stage resource management**, that jointly with a novel synchronization primitive, **bulk semaphores**, enables executing the maximum number of concurrent allocation/free operations without running into the *false resource starvation* problem. Many components of our GPU memory allocator will rely on these techniques to offer high allocation throughput.

3.2 Two-stage Resource Management

Counting semaphores [10] provide a natural way to track resources and solve the *false resource starvation* problem. For instance, given a fixed amount of a resource, we track how many units are left using a counting semaphore initialized to the initial number of resource units. A counting semaphore keeps an internal integer value, S, and supports two operations:

- `wait(N)`: if $S \geq N$, then $S \leftarrow S - N$; otherwise, the calling thread is blocked.
- `signal(N)`: $S \leftarrow S + N$, and wake up the waiting threads to re-attempt the `wait` operation.

Threads execute a `wait` operation to acquire an integer number of units of resource, and a `signal` operation to release them.

If the amount of a given resource can grow and shrink over time, we can modify the semantics of `wait(N)` to support this use-case.

- if $S \geq N$, then $S \leftarrow S - N$ and return N.
- if $N > S \geq 0$, then $S \leftarrow -1$ and return S.
- otherwise, the calling thread is blocked.

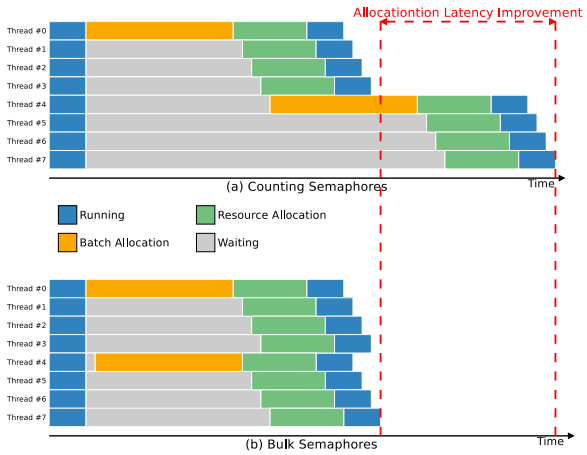


Figure 1. Two-stage allocation example using (a) counting semaphores, and (b) bulk semaphores.In the example 8 concurrent threads use two-stage allocation with a resource batch size of four units.

If `wait` returns a value smaller than the requested amount of resources, N, the calling thread is in charge of growing the resource pool. For instance, consider the example in Figure 1(a), where the semaphore count is initially 0. The first thread calling `wait` (Thread #0 in the figure), sets its value to -1 and receives 0 as a return value. Since the return value, 0, is smaller than the amount of resources requested, 1, the thread starts allocating a new batch of resources (four in our example). All other threads block on the `wait` operation. Once Thread #0 allocates the new batch, it executes a `signal(4)` operation and wakes up Threads #1-4. Notice that Thread #4 gets 0 as return value, so it allocates a new batch.

Counting semaphores use a two-stage resource management scheme. In the first stage, a call to `wait` allocates a given amount of resources. On a second stage, a lookup and update operation over some data structure, e.g., a bitmap, locates which resources were allocated. Notice that the number of threads concurrently accessing the tracking data structure is limited to the number of available elements, which limits the contention overheads [5].

Two-stage resource management using counting semaphores has a built-in scalability barrier. Once a thread is allocating a new batch of resources, all arriving threads block. If the number of concurrent threads is large, by the time the resources in the new batch become available, it is very likely that the number of waiting threads is larger than the batch size [13]. The strategy of increasing the bach size proves inefficient, since we have to keep increasing it as the number of concurrent threads grows, leading to unacceptable resource fragmentation.

Algorithm 1 `WAIT` operation on a batch semaphore

```
procedure WAIT(Sem, N, B)
    while True do
        atomic
            if Sem.C + Sem.E − Sem.R < N then
                Sem.E ← Sem.E + B − N
                return −1
            else if Sem.C ≥ I then
                Sem.C ← Sem.C − N
                return 0
            else
                Sem.R ← Sem.R + N
            end if
        end atomic

        while Sem.C < N and Sem.R < (Sem.C + Sem.E) do
            yield
        end while

        atomic
            Sem.R ← Sem.R − N
        end atomic
    end while
end procedure
```

3.3 Bulk Semaphores

We develop *bulk semaphores* to enable scalable two-stage resource management. A batch semaphore is based on three counters:

- C: the semaphore value, e.g., how many units of a resource are currently available.
- E: the expected counter, e.g., how many units of a resource we expect to become available.
- R: the reservation counter, e.g., how many units of a resource have been reserved by waiting threads.

We define *expected availability* of a bulk semaphore as its value after all expected units, E, have been added to the semaphore, and all reserved units, R, have been subtracted. This quantity determines if a thread can eventually decrement N units from the semaphore without resulting in a negative value. A thread trying to decrement the semaphore by a quantity N smaller than its expected availability can safely wait until $C \geq N$. Otherwise, the thread would need to allocate a new batch of units for the semaphore.

We extend the semantics for `wait` and `signal` operations in bulk semaphores. Besides the amount N to increment/decrement the semaphore value, these operations also take a batch size, B, as input parameters. Algorithm 1 describes a `wait` operation on a batch semaphore. If the semaphore expected availability is less than the decrement value, it increments the expected value E by the batch size B. Otherwise, if the semaphore counter is greater or equal than the decrement value, it decrements the semaphore counter C. Otherwise, the code increments the reservation counter R by N and waits until any of the previous conditions are met.

Algorithm 2 describes a `signal` operation on a batch semaphore. This operation takes two integers N and B as inputs, increments the semaphore value by the sum of the inputs $(I + B)$, and decrements the expected counter E by B. If the integer B is different from zero, it wakes up to B threads

Algorithm 2 SIGNAL operation on a batch semaphore

```
procedure SIGNAL(Sem, N, B)
    atomic
        Sem.C ← Sem.C + N
        Sem.E ← Sem.E − B
    end atomic
end procedure
```

blocked in a `wait` operation. Notice that bulk semaphores generalize counting semaphores. If the expected count is infinite, `wait` and `signal` operations have the same semantics as in counting semaphores as long as we use a zero value for the batch size B.

When the return value of a `wait` operation is non-zero, the thread will eventually add the promised units to the semaphore using a `signal` operation. If the thread fails to add the promised elements, it must also signal the condition to the semaphore. Both, `wait` and `signal` operations must update the value of several counters atomically. A possible implementation packs all the semaphore counters into a 64-bit word, and uses atomic *compare-and-swap* operations to update them.

Figure 1(b) illustrates how bulk semaphores enable scalable two-stage allocations. Thread #0 executes a `wait(1, 4)` operation, which sets the expected count to 3, and returns −1. Thread #0 starts allocating a new batch of resources while Threads #1-#3 execute `wait` operations over the semaphore, which increment the reserved count from zero to three. When Thread #4 executes a `wait` operation, all expected resources have been reserved and, thus, the thread increments the expected count, and starts allocating a new batch of resources. This is in contrast to counting semaphores in Figure 1(a), where Thread #4 remains blocked until Thread #0 signals the semaphore.

Threads block on a `wait` operation whenever there are enough expected resources to supply them. If more resources are needed to meet demand, bulk semaphores let many concurrent threads allocate as many new batches as are actually needed. The perfect match between the number of waiting threads and the number of expected resources provides an efficient solution to the *false resource starvation* problem.

4 GPU Memory Allocator

Our allocator exposes its functionality to the application through the standard `malloc` and `free` interfaces. For `malloc`, we first round the allocation size up to the closest power of two, and forward the request to a fine-grained allocator, *UAlloc*, or to a coarse-grained allocator, *TBuddy*, depending on the allocation size.

A key property of our design is that memory allocated by *TBuddy* is guaranteed to be page aligned, while memory allocated by *UAlloc* is guaranteed to never be page aligned. We exploit these guarantees on calls to `free`, inspecting the memory address and forwarding it to *TBuddy* if the address is page aligned or to *UAlloc* otherwise. This removes the need for a shared data structure to track the ownership of allocations, which might become a potential point of contention in our design.

4.1 The Tree Buddy Allocator

Most CPU user-level allocators forward coarse-grained allocation requests to the OS kernel, such as via `mmap` in POSIX. The OS kernel allocates a contiguous range in the process virtual address space using a simple algorithm (e.g., first-fit) that delivers high performance at the cost of high fragmentation. OS allocations tolerate high memory fragmentation because the size of the virtual address space (e.g., 256TB for x86-64 CPUs) is orders of magnitude larger than the amount of physical memory. A GPU memory allocator can only allocate memory within a small memory pool pre-allocated by the CPU. Hence, we implement coarse-grained allocations using a buddy system, which offers a good trade-off between performance and memory fragmentation [7, 11, 14].

Buddy systems are typically implemented as a table of free-lists. The index of a table entry corresponds to the allocation *order* for the blocks in the free-list, where the size of each block is $Page_{size} \times 2^{order}$.

An allocation operation of order n, removes the first element of the n^{th} free-list. If the list is empty, the system allocates a block of order $n+1$ and splits it into two blocks of order n—these two blocks are called *buddies*. The allocator inserts one block in the free-list of order n, and marks the other block as busy. A free operation for a block of order n checks if the buddy block is available. If it is, the allocator marks the buddy block as busy, removes it from the free-list, merges both buddies into a block of order $n+1$ and executes a free operation over this new block. If the buddy is busy, the allocator marks the block as available, and inserts it into the free-list.

Allocation and free operations lock the buddy system to prevent concurrent accesses to the free-lists and avoid *false resource starvation*. The performance of this implementation degrades as we increase the number of concurrent operations due to the serialization introduced by its use of locking. Our goal is to design a buddy system that enables concurrent allocation and free operations while avoiding false resource starvation.

Figure 2 shows the design of our Tree Buddy (*TBuddy*) allocator. We use *two-stage resource management* (Section 3.2) to track memory blocks of each order. One batch semaphore per order, with batch size of two, tracks the number of available blocks. A static binary tree tracks the state of memory blocks. A node of height h tracks a memory block of order h, and can be in three different states: **Available**, if the memory block can be allocated; **Busy**, if the memory block and all the memory blocks in its sub-tree cannot be allocated; or **Partial**, if the memory block cannot be allocated, but its sub-tree contains at least one **available** block.

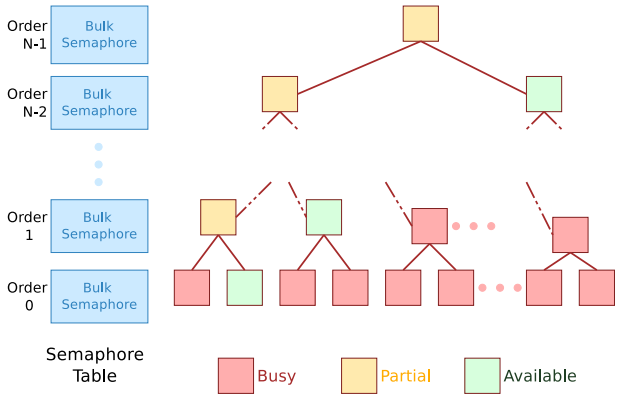


Figure 2. Design of our coarse-grained Tree Buddy Allocator

Notice that for the tree to describe a buddy system, the following two properties must hold. First, two sibling nodes cannot be **available** at the same time. Any operation that results in this state must set both nodes as **busy** and their parent node as **available**. Second, if a node is **available**, all the nodes in its sub-tree must be **busy**. This ensures that all allocations happen over disjoint memory ranges.

An allocation operation of order n, first waits on the n^{th} batch semaphore. If the result of this operation is 0, it traverses the tree to locate a node of height n in the **available** state, and atomically switches it to **busy**. If the result of `wait` is -1, it allocates a block of order $n+1$, and splits it by switching the block state from **busy** to **partial**, and the state of one child node from **busy** to **available**. Finally, the thread signals the semaphore to notify the existence of a new available block.

A free operation of order n first attempts to merge the block with its buddy. If the merge operation fails, the thread switches the block from **busy** to **available**, and signals the semaphore. A merge operation only proceeds if the buddy of the block triggering the operation can be allocated. To support this, we augment batch semaphores with a `try-wait` operation, that decrements the semaphore if its value is positive, and returns an error otherwise. If the `try-wait` operation succeeds, the merge operation executes an atomic compare-and-swap operation to transition the buddy block from **available** to **busy**. If the state transition fails, the thread signals the semaphore and returns a failure. Otherwise, the parent block transitions from **partial** to **available**, and the thread *signals* the $(n+1)^{th}$ semaphore. A free operation must attempt a merge even if the buddy block is **busy** because a concurrent free operation might release it in the meantime. Only the failure to decrement the batch semaphore guarantees that the merge cannot proceed.

To ensure the consistency of the buddy tree, we propagate allocation and free operations from the node to its parent. If a node transitions from **available** to **busy**, the parent node might need to transition from **partial** to **busy**. If a node transitions from **busy** to **available**, the parent node might need to transition from **busy** to **partial**. Analogously, transitions from **busy** to **partial**, or from **partial** to **busy** might require the parent node to also perform the same transition. During the propagation process we might run into race conditions if two threads attempt to transition the state of the same node. For instance, let us assume a thread has freed a node (**busy** → **available**), whose parent is in a **busy** state. Before this threads propagates the state change to the parent node, a concurrent thread allocates the very same block (**available** → **busy**). When the first thread propagates the free operation to the parent, it forces a transition to an inconsistent state (**busy** → **partial**). To prevent race conditions we lock the node and its parent before performing any state transition. Notice that if we need to update the grandparent node, we only need to hold the lock for the parent and the grandparent node.

4.2 The UnAligned Allocator

The Unaligned Allocator (*UAlloc*) resembles existing concurrent CPU memory allocators. Figure 3 shows how memory is organized in *UAlloc*. We assign one arena to each SM, i.e., up to 2,048 threads can be sharing an arena to maximize the L1 cache hit rate of applications [4]. Each arena handles memory in chunks of 512KB which are further sub-divided into 4KB bins. Once initialized, a bin contains allocations of a fixed size. A bin has a 128 byte header that includes a bitmap to track up to 512 different allocations; the minimum allocation size is therefore 8 bytes. If the bin size is larger than 8 bytes, the allocator initializes the bitmap to allow allocating only the number of available blocks. The arena keeps a per-size bin free-list to store bins with available elements.

The first two bins in a chunk are special. The first bin contains the chunk header that includes a bitmap to track the state of the 64 bins in the chunk. The bottom 3,968 bytes of these two chunks are divided into 62 tails, 128 bytes each. The allocator logically appends each tail to the end of each bin, so the total number of bytes available to allocate in a bin is effectively 4KB. If the allocation size is larger than 128 bytes, the allocator cannot use the *tail space*. However, for smaller allocations, this design reduces the allocator internal fragmentation minimizing the amount of wasted space.

Since *TBuddy* always returns allocations aligned to the allocation size, each chunk is aligned to 512KB, and each bin is therefore 4KB aligned. Given that the first 128 bytes of each bin are always used to keep the allocator internal data, any allocation from *UAlloc* is guaranteed to never be aligned to 4KB. As previously discussed, this enables determining whether an allocation belongs to *UAlloc* or to *TBuddy* by simply inspecting its alignment.

We implement allocation operations in *UAlloc* using two-stage resource management. The first stage executes a `wait` operation over the batch semaphore associated with the bin free-list for the allocation size. If this operation returns zero, we allocate a new bin from any of the chunks in the arena

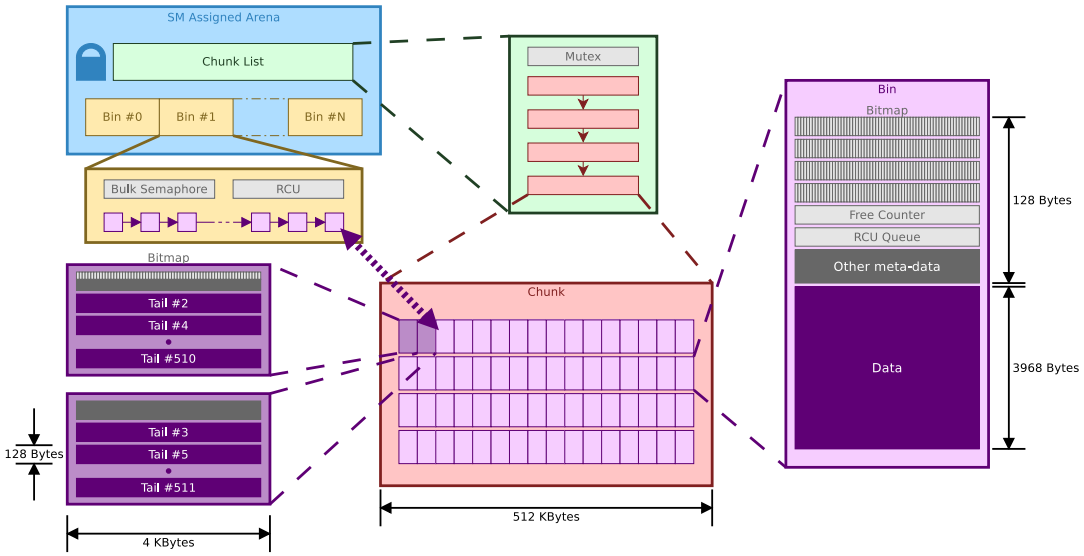


Figure 3. Design of our fine-grained UnAligned Allocator

chunk list. If this list is empty, i.e., there are no chunks with available bins, we call *TBuddy* to allocate a new chunk. The second stage traverses the bin free-list until it manages to decrement a bin free counter without turning it negative. Finally, we inspect the bin bitmap for a zero bit, and we atomically set it to one. From the index of this bit, we compute the offset of the allocated memory block within the chunk or its associated tail. Bin allocations use a analogous approach using the chunk bitmap.

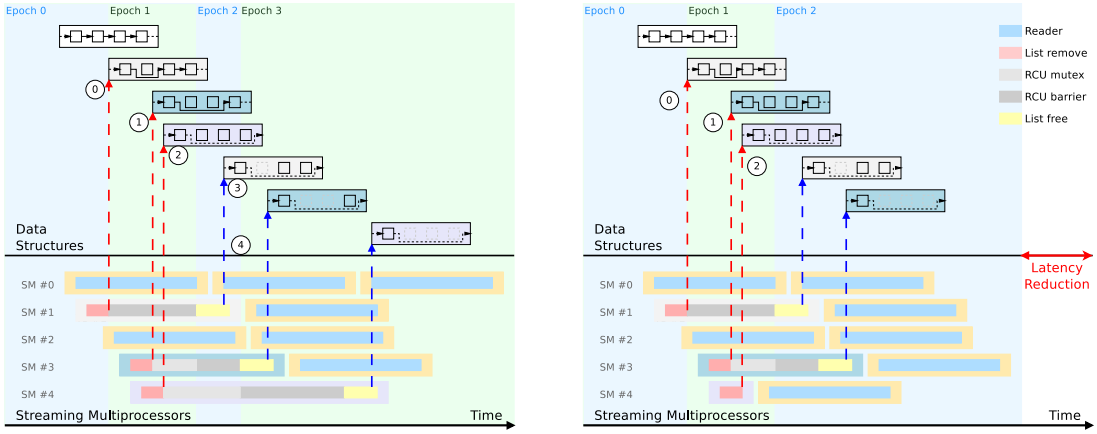
4.2.1 Delegated Execution in Read-Copy-Update

An allocation operation might exhaust a bin, meaning that all its blocks become used. In this case, the allocation removes the bin from the free-list. Analogously, a free operation inserts a bin back into the free-list if the bin was exhausted, and removes it from the free-list when freeing the last memory block. Since many threads can be (de)allocating memory concurrently, we need to enclose the list traversal within a critical section. We expect few threads to actually update the list, while the majority only update the bin bitmap. Hence, we use Read-Copy-Update [18] (RCU) as our synchronization mechanism. Figure 4a illustrates how RCU manages concurrent accesses to a linked-list. RCU allows readers (light blue bars in Figure 4a) to access the list even if one thread is updating it. Writer threads execute serially (⓪, ①, and ② in Figure 4a) through a separate synchronization mechanism, e.g., a mutex. A writer thread first logically updates the list, e.g., unlinks one element from the list (⓪ in Figure 4a), and adds a callback to actually release the element to the RCU queue. Then the writer issues an RCU barrier to wait for all readers to exit the RCU critical section, before executing the callbacks in the RCU queue (③ in Figure 4a).

Most existing user-level RCU implementations [9] use per-thread variables, and RCU operations require iterating over all those variables. Given the large number of threads in a GPU, these implementations incur very large overheads. Sleepable Read-Copy-Update (SRCU) [17], originally designed to allow reader threads to block within RCU read-side critical sections, provides a suitable GPU implementation. SRCU keeps an *epoch counter* and a pair of *reader counters*, which track the number of readers in two consecutive epochs. Reader threads increment the reader counter of the current epoch upon entering the RCU read-side critical section and decrement its value upon exit. An RCU barrier acquires the RCU mutex, increments the epoch counter (colored stripes in Figure 4a), waits for the RCU counter for the previous epoch to be zero (i.e., all readers exited the RCU critical section), and releases the RCU mutex.

Notice that in SRCU barriers wait for all the threads that entered an RCU read-side critical section before the epoch counter is incremented, even if they did so after the RCU barrier was issued. Although functionally correct, this can degrade the application performance. For instance, in Figure 4a the RCU barrier issued at ② cannot update the epoch counter until the RCU barrier issued at ① releases the RCU mutex at ④. Therefore, this RCU barrier waits for reader threads that started executing long after the barrier was issued ③, occupying hardware resources and preventing other thread-blocks from being executed. The large number of concurrent threads in a GPU causes this effect to happen quite often, resulting in significant performance overheads.

A property of SRCU is that if a RCU barrier (e.g., ② in Figure 4a) is issued while another RCU barrier is waiting to increment the epoch counter (e.g., ① in Figure 4a), both



(a) Traditional RCU **(b)** GPU-aware RCU

Figure 4. RCU example running on a GPU using a linked-list. All thread-blocks include threads traversing the list, and thread-blocks #0, #2, and #3 include one thread removing one element. The top part shows the state of the linked-list as it is updated, the middle part shows the timeline for thread execution, and the bottom part shows a possible mapping between thread-blocks and SMs in the GPU.

barriers can clear at the same time. We can exploit this to minimize the time an SM is occupied waiting for RCU barriers to clear by defining a conditional RCU barrier. A conditional RCU barrier returns immediately when another RCU barrier is waiting to increment the epoch counter, e.g., ② in Figure 4b. Otherwise, it becomes a full RCU barrier, e.g., ⓪ and ① in Figure 4b. This construct delegates the execution of RCU callbacks to the thread waiting on the RCU barrier, hastening the release of hardware resources.

4.2.2 Collective Synchronization Primitives

Several threads within a thread-block often need to allocate a new chunk concurrently. Prior to removing a chunk from the list, threads lock the list mutex to prevent race conditions, serializing their execution.

We exploit the concurrency guarantee of thread-blocks in CUDA to reduce the serialization of chunk allocations. We protect each chunk list using a **collective mutex**. A collective mutex allows a set of threads, e.g., a thread-block, to collectively lock and unlock the mutex. Upon locking the mutex, all threads in the thread-block enter the critical section. Within the critical section, the threads can coordinate with each other using barriers as needed. Each individual thread executes an unlock operation over the mutex when it leaves the critical section, but the mutex remains locked until all threads in the thread-block have executed the unlock operation. For instance, during a chunk allocation, each thread computes its index within the group of threads performing the operation, collectively enters the critical-section, and traverses the number of elements dictated by its index, to locate the chunk to allocate. Before leaving the critical-section, the thread with the highest index, splits the list by the chunk it is allocating. This implementation allocates several chunks with a single list operation, whereas the code using a regular mutex requires as many list operations as allocations.

Although we use mutexes to illustrate the concept of collective synchronization primitives, we can easily generalize this concept to expand to other constructs. During an **collective acquire** operation one thread is selected to actually acquire the synchronization object, e.g., `lock` a mutex. A barrier at the end of the implementation ensures that the operation does not return until the elected thread has effectively acquire the synchronization object. Analogously, a **collective release** operation first requires all participating thread to arrive to a barrier, and then a elected thread effectively releases the synchronization object.

5 Evaluation

We evaluate our synchronization primitives and memory allocator using the CUDA Toolkit version 9.2 and an NVIDIA Titan V GPU. For each thread count, we run kernels using thread-blocks of power-of-two (from 1 to 1,024) number of threads and average the execution time for all of them.

5.1 Bulk Semaphores

We compare the allocation throughput of counting and bulk semaphores when using two-stage resource allocation assuming we can allocate a batch of resources using a single atomic operation. In this benchmark each thread allocates one unit of resource from a batch, and batches are allocated as they become empty. This experiment factors out the effect of the resource allocator and gives us an upper limit to the achievable allocation rate for each synchronization primitive.

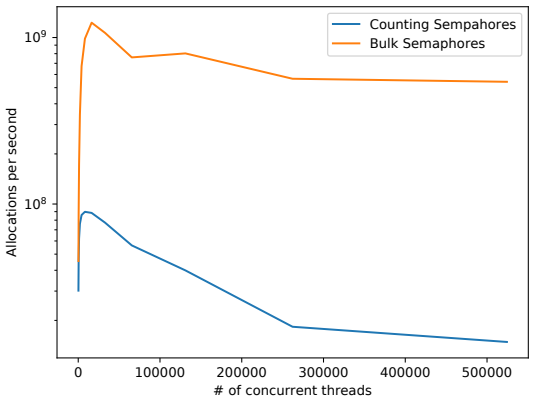


Figure 5. Upper limit allocation throughput using counting and bulk semaphores

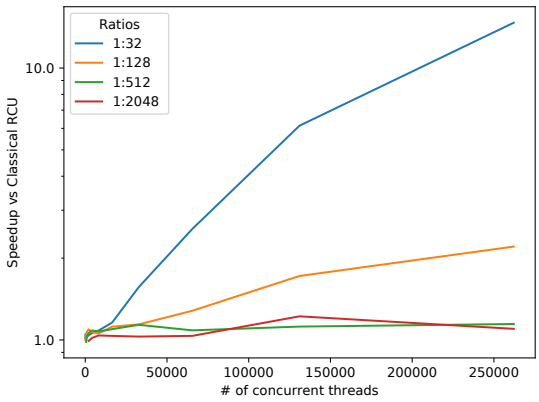


Figure 6. Speedup of RCU Delegation vs. Classical RCU for different ratios of writer:reader threads

We run this experiment for different batch sizes and number of concurrent threads. Figure 5 shows the allocation rate for a batch size of 512, which matches the batch size we use in *UAlloc*. Experimental results for other batch sizes are analogous to Figure 5. As we hypothesized in Section 3.3, bulk semaphores outperform counting semaphores [21] due to concurrent batch allocations.

We observe that the allocation throughput starts decreasing before the number of concurrent requests peaks. Our profiling data indicates that this is an artifact of our bulk semaphore implementation. In our implementation, updates to the reserved value (R) saturate the atomic throughput of the GPU sooner than the updates to the semaphore value (E). The atomic throughput limits the maximum number of concurrent batch allocations, which never reaches its theoretical maximum value, i.e., number of requests over the batch size.

5.2 RCU Delegation

We evaluate RCU delegation using a benchmark that removes those elements in a double linked-list whose tag matches values in an input vector. In this benchmark, each GPU thread traverses the list searching for one input tag, and uses RCU to enable concurrent traversals and removals. To control the ratio between reader and writer threads, we ensure that the input tag vector contains all tags in the list. The number of tags corresponds to the number of readers, while the number of elements in the list matches the number of writers.

Figure 6 shows the speedup of RCU delegation versus classical RCU on this benchmark for different ratios of reader and writer threads. In the worst-case, RCU delegation is 1.01% slower than classical RCU. For small thread counts, both implementations deliver similar performance; our experimental data shows that RCU delegation is 2.3% faster than classical RCU, which is within the measurement error. As expected, delegation provides minor performance gains for small thread counts since the chance of two thread-blocks waiting for the same set of reader threads is small. Analogously, few collisions happen when the number of writers is very small compared to the number of readers. However, as the number of writers increase, whether by increasing the number of concurrent threads or increasing the ratio of writers to readers, RCU delegation prevents unnecessary occupying hardware resources, speeding up the execution by as much as 14X on this benchmark.

5.3 Allocator Performance

Having examined the performance of key concurrency primitives used by our allocator, we now turn to the performance of the allocator itself. We focus specifically on total allocator throughput, which we characterize in terms of allocation requests completed per second.

We use the standard CUDA system allocator—available via the `malloc` and `free` interfaces in device code—as the baseline for comparison. While the HAlloc allocator demonstrated better performance for select benchmarks [3] on the Kepler architecture, we found that it crashes in most experimental cases on the Volta architecture. This appears to be the result of unsafe warp-synchronous programming idioms that result in data races on the newer architecture. As a result, we are unable to include comparative data on its performance.

To measure allocation throughput, we run a benchmark where each thread performs a single call to `malloc` to request a fixed amount of memory. We run just enough threads to completely exhaust the memory pool, so that no memory remains free once all threads have finished. By exhausting the memory pool, we can indirectly measure fragmentation based on the number of threads that fail to allocate memory,

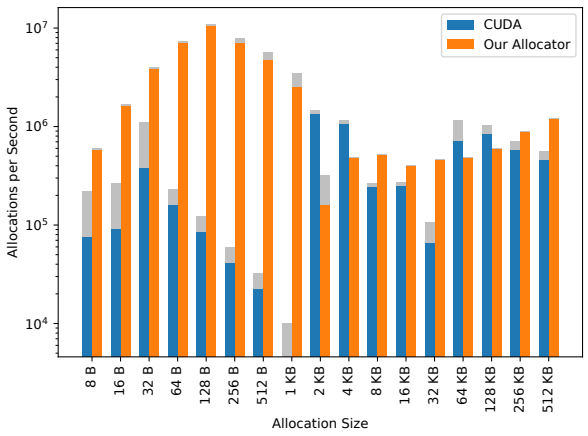


Figure 7. Allocation throughput for CUDA and our allocator for different allocation sizes

since there would be no failures in the absence of fragmentation. This also ensures that we measure allocator performance for the full spectrum of resource availability, from the case where the entire pool is empty to the case where precisely one piece of memory remains.

We repeat this benchmark for a range of allocation sizes. For the smallest allocation size of 8 bytes, we use an 8MB memory pool (i.e., we run 2^{20} threads). We increase the memory pool as we increase the allocation size to keep running one million threads for each case until we reach a memory pool size of 512MB for a 512-byte allocation size. After this size, we reduce the number of allocations keeping a 512MB memory pool. Notice that this changes the contention profile for sizes larger than 512 bytes, e.g., only 1,024 threads run when allocating 512KB blocks. We use this memory pool sizing strategy to keep the total execution time of our benchmarks within reasonable limits. For example, exhausting a 512MB memory pool using 8 byte allocations using all possible configurations takes well over 72 hours.

Figure 7 shows the allocation rate for CUDA's standard allocator and our allocator. On each configuration, the gray top bar indicates the contribution of failed allocations to the allocation rate. Our allocator delivers a higher performance for all allocation sizes expect for 2KB, 4KB, 64KB, and 128KB. Besides the low allocation rate, the 2KB case also shows a high number (50%) of failed allocations. This is a degenerated case in our allocator since it actually rounds them up to 4KB, since a bin cannot hold two 2KB memory blocks. All the allocation sizes where our allocator is outperformed by the CUDA allocator are handled by the buddy system. As expected from our design, the buddy system delivers a mostly constant allocation rate, which increases for larger allocation sizes due to the smaller thread count. This is not the case for the CUDA allocator, which presents two allocation rate peaks for the sizes where it outperforms our allocator.

In Figure 7 we also observe that our allocator presents a much lower failure rate. Our allocator presents a small number of failures for the allocations handled by *UAlloc* due to the memory used for the chunks and bins headers. For allocation sizes larger than 4KB, handled by the buddy system, our allocator never fails allocating. Besides the degenerate case for 2KB, our allocator also has a moderate failure rate for 1KB and 512B allocations. This is a direct consequence of the fixed bin size used in our design; from a 4KB bin devoted to 1KB allocations, only 3KB are available for allocations. These results indicate that we should use different bin sizes for small and large allocations, even though this design change is not likely to improve the allocation throughput.

6 Conclusions

The massive number of concurrent threads in modern GPUs pushes traditional synchronization primitives beyond their scalability limits. We have experienced this while designing a throughput-oriented memory allocator for NVIDIA GPUs. In this paper, we have explored how we can adapt existing techniques and primitives to match the scalability requirements of modern GPUs.

Our experience shows that existing design patterns, e.g., existing concurrent CPU memory allocators, can be adapted to the concurrency levels of GPUs. However, such designs tend to include synchronization primitives, e.g., semaphores, whose effects are acceptable for the thread counts of existing multi-processor systems, but become major bottle-necks when the number of threads is three orders of magnitude larger. We found it necessary to design new synchronization primitives, e.g., bulk semaphores, that avoid serialization to achieve high performance. Our experience also indicates that rather than building completely new synchronization constructs, we can adapt existing ones to fit the requirements of GPU execution. Finally, we have shown how these low-level techniques can be used to design a dynamic memory allocator that outperforms the standard implementation provided by CUDA by as much as two orders of magnitude in our benchmarks.

References

[1] [n. d.]. Kinetica Web Page. https://www.kinetica.com/a. Accessed: 2018-12-30.
[2] [n. d.]. RAPIDS Web Page. https://rapids.ai/. Accessed: 2018-12-30.
[3] Andrew V Adinetz and Dirk Pleiter. 2014. Halloc: a high-throughput dynamic memory allocator for GPGPU architectures. In *GPU Technology Conference (GTC)*. 152.
[4] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. 2000. Hoard: A scalable memory allocator for multithreaded applications. In *ACM SIGARCH Computer Architecture News*, Vol. 28. ACM, 117–128.
[5] André B Bondi. 2000. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*. ACM, 195–203.
[6] Jeff Bonwick and Jonathan Adams. 2001. Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources..

In *USENIX Annual Technical Conference, General Track*. 15–33.

[7] Daniel Bovet and Marco Cesati. 2005. *Understanding The Linux Kernel*. Oreilly & Associates Inc.

[8] NVIDIA Cooprporation. [n. d.]. *cuSolver Manual*. NVIDIA.

[9] Mathieu Desnoyers, Paul E McKenney, Alan S Stern, Michel R Dagenais, and Jonathan Walpole. 2012. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems* 23, 2 (2012), 375–382.

[10] Edsger W Dijkstra. 1968. Cooperating sequential processes. In *The origin of concurrent programming*. Springer, 65–138.

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

[12] Xiaohuang Huang, Christopher I Rodrigues, Stephen Jones, Ian Buck, and Wen-mei Hwu. 2010. Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. IEEE, 1134–1139.

[13] Leonard Kleinrock. 1976. *Queueing systems, volume 2: Computer applications*. Vol. 66. wiley New York.

[14] Kenneth Knowlton. 1965. A Fast Storage Allocator. *Commun. ACM* 8, 10 (1965), 623–624.

[15] Alex Kogan and Erez Petrank. 2011. Wait-free queues with multiple enqueuers and dequeuers. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 223–234.

[16] Romolo Marotta, Mauro Ianni, Andrea Scarselli, Alessandro Pellegrini, and Francesco Quaglia. 2018. A non-blocking buddy system for scalable memory allocation on multi-core machines. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 164–165.

[17] Paul E McKenney. 2006. Sleepable Read-Copy Update.

[18] 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*. 509–518.

[19] NVIDIA Coorporation. 2017. *NVIDIA Tesla V100 GPU Architecture*. Technical Report WP-08608. http://www.nvidia.com/object/volta-architecture-whitepaper.html

[20] 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, 1–10.

[21] Jeff A Stuart and John D Owens. 2011. Efficient synchronization primitives for GPUs. *arXiv preprint arXiv:1110.4623* (2011).

[22] M. Vinkler and V. Havran. 2014. Register Efficient Memory Allocator for GPUs. In *Proceedings of High Performance Graphics (HPG '14)*. Eurographics Association, Goslar Germany, Germany, 19–28.

[23] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 11.

[24] 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*. ACM, 120–126.