Throughput-Oriented GPU Memory Allocation

Ansh Rupani

May 14, 2019

Abstract

There has been an ever growing demand of multi-core architectures for achieving higher levels of concurrency. Multi-threaded programing faces new challenges on throughput-oriented architectures like GPUs. The immense performance pressure on synchronization primitives like mutexes on tasks like memory allocation is unimaginable due to the extremely high number of threads in such architectures. The authors in [GG19] propose various concurrent programming techniques for GPU architectures to support better dynamic memory allocation.

The authors have build their memory allocator with the support of other improvements they have proposed in the paper. They have proposed bulk semaphores, a generalized variant of counting semaphores which supports the case when multiple threads operate on a semaphore concurrently. A better approach for synchronization mechanisms like Read-Copy-Update has been proposed where multiple threads are allowed to access the critical section of the program simultaneously. Also, techniques to delegate deferred reclamation to threads which are already blocked have been proposed. All these proposed techniques when combined together contribute towards a highly throughput-oriented GPU memory allocator, which performs 16.56 times better than the CUDA based memory allocator in terms of allocation rates.

Contents

1 Introduction

The quest for fast computing is ever growing. Researchers and industrialists have tried to live up to the expectations of the newer technologies by increasing the number of processing cores in the system. This hardware solution for providing speed up to the softwares was a gradual process, which ultimately reached its limits. However, the speed-up, as presented by the Amdahl's law, is limited by the serialized section of the software. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20 times.

Graphical Processing Units however, are capable of providing more concurrency as compared to the normal multicore architectures. But using their capability to the fullest has however remained a debatable question. Various problems have to be targeted when utilizing the power of concurrency provided by the GPUs. One of the most important problem is of memory allocation.

Memory allocation arises as a problem in multi-threaded programming when the number of competing threads increases. The use of the common synchronization primitives like mutexes and semaphores becomes a bottleneck. This happens because a lot of time is spent in locking and unlocking of mutexes. This time increases with increase in the level of concurrency. Further, the problem of memory fragmentation can be involved in such cases. False resource starvation is another important issue to take care of while using platforms like GPUs.

Therefore, some improvisation is required when employing the concurrency offered by high performing GPUs for any software. The paper targets highly concurrent platforms such as GPUs and proposes concurrent programing techniques and synchronization primitives in support of a dynamic memory allocator.

2 Background

2.1 Synchronization Primitives

Synchronization primitives are an essential part of concurrent programming in general. These primitives help to provide multithreading semantics to the kernel. They are some software mechanisms exposed over platforms like the OS to the users for synchronization of threads or processes. They are implemented by various low level mechanisms such as atomic operations and memory barriers. Some of the well known synchronization primitives are mutexes, semaphores, conditional variables and shared or exclusive locks. *Mutexes* are primitives which provide mutual exclusion for one or more data objects. It is basically a locking mechanism, which allows multiple threads to access a particular resource but not at the same time. The lock can only be released by the thread that has acquired it. *Semaphores*, on the other hand, are signalling mechanisms. They allow multiple threads to access a limited number of resources. The semaphore value in general can be modified by any thread acquiring or releasing the resource. In *counting semaphores*, for instance, given a fixed number of resources, the users track the number of resources remaining by employing a counting semaphore initialized to the initial number of total resources available. There are two operations associated with counting semaphores: wait(N) and signal(N). 'N' is the number of resource units requested by a thread and 'S' is the number of total resources available at a particular time. While executing the wait(N) operation, if $S >= N$, $S = S - N$(or else the callee thread is blocked). While performing signal(N), $S = S + N$, and the waiting threads are signalled to perform the wait(N) operation.

2.2 Dynamic Memory Allocating Process

The dynamic allocation of memory is performed with the help of the OS and the memory allocator employed by the user. The virtual address space is allocated by the OS and the physical pages are committed as needed to provide storage resources. Then the the task falls upon the user level memory allocators, through interfaces like *malloc*. These interfaces sub-divide the memory spaces provided by the OS as required by the applications upon their requests. Within these allocators, memory is managed in fixed sized *chunks*, by *arenas*. Arenas also define a set *bins*, which individually correspond to a fixed allocation size. They also include metadata to efficiently locate available memory blocks of that particular size. Synchronization primitives like mutexes are used to protect arenas to support concurrent allocation and deallocation of memory.

2.3 Multi-threaded Programming in GPUs

CUDA is used as the concurrent programming paradigm for GPUs. CUDA applications execute GPU tasks in grids, which further organize kernel threads into thread blocks.Within a thread block, all threads share the access to the on chip memory and they can synchronize with each other using the hardware supported barriers. A particular thread block is associated with a Streaming Multiprocessor, which is the hardware unit on which all the threads of the thread block will be executed. Individual threads that run on the GPU interface with malloc for dynamic memory allocation.

3 Motivation

As mentioned earlier, the paradigm of concurrent programing requires special focus when performed on GPUs. The latest GPUs are capable of exhibiting upto 172,032 threads, as compared to the lower number exhibited by the CPUs. With this enormous thread population, challenges are exhibited by the applications which rely on concurrent data structures and employ the use of synchronization primitives to check the access to these data structures. This is so because a lot of time is spent in locking and unlocking of synchronization primitives like mutexes. This time increases with the increase in the number of threads and overpowers the benefit of multi-threaded programming. Therefore, the task of *dynamic memory allocation* is difficult in such systems.

Applications like graph and data analytics, sparse linear algebra etc. use dynamic memory. But often, programmers tend to solve the above stated issue with the help of an upper bound array. However, this results in memory wastage and also limits the dataset size for the application.

Another problem of *memory fragmentation* is also observed in such highly-concurrent systems. If left unchecked, memory fragmentation can grow enormously at high allocation rates. Therefore, it is necessary to minimize it.

Further, issues like *false resource starvation* are noticed, which are however not related to the concurrency levels of GPUs. We take the example of a wait free queue to demonstrate this prbolem, which occurs by the use of resource allocators like *jemalloc* in general. Suppose the reource pool, here, a wait-free queue, contains all the available memory in a single memory block. One thread performs the deque operation to allocate the requested amount of memory. While this is being doe, the queue is locked. At the same time, if another thread tries to request some memory, it would find the memory ersource pool empty, because it has been reserved for a previous operation, which in turn does not request the entire block. Problems like this need to be solved in an intelligent way to provide high-throughput memory allocation for GPUs.

It is necessary to adapt existing synchronization techniques and primitives to match the scalability criterias of the modern day GPUs.

4 Previous and Related Work

The authors in [GG19] target the domain of dynamic memory allocation specifically for massively concurrent architectures like GPUs. However, there already has been some work in this domain of research.

ScatterAlloc [SKKS12] is a massively parallel dynamic allocator for the GPU. The authors focus on reducing collisions and congestions by scattering incoming memory requests by using techniques like hashing. ScatterAlloc has been shown to yield speedups between 10 to 100 as compared to the CUDA toolkit. It is focused at optimizing CPU level techniques for the GPU. They have pointed out issues such as memory access performance, scalability, diverging execution paths, false sharing, coalesced access, internal fragmentation, external fragmentation and blowup. It tracks memory availability using bitmaps. To start with, a parallel list-based allocator has been used as the baseline, based on the first-fit algorithm. ScatterAlloc scatters allocation requests across memory blocks of fixed sizes, while trading in terms of fragmentation for achieving allocation speed, to avoid collisions in requests. ScatterAlloc splits memory into pages. A page usage table is used to keep track of the free memory. Further, pages are grouped in super blocks. For speeding up the search for available memory, metadata is stored in each super block. The present work [GG19] also uses a similar technique which involves scattering of the traversal of a static binary tree in the coarse-grained allocator to prevent collisions while locating the memory blocks.

Further, the authors in [WWWG13] propose to solve the problem of dynamic memory allocation in architectures like GPU by utilizing the SIMD parallelism observed in almost all massively parallel hardware architectures, with the help of FDGMalloc. They propose a dynamic memory allocator which operates at the SIMD parallel warp level, as far as CUDA is concerned. The work is based on the key observation that architectures like GPUs operate in an SIMD fashion where a single instruction is executed in parallel physically. Memory management tools should take this fact into account to obtain speed ups while functioning at such level of granularity. In this approach, simultaneous requests which are quite frequent are handled very efficiently with the help of a voting function along with the use of fast allocation within a super block of memory. However, there allocator relies on the CUDA allocator. This work ultimately targets frequent dynamic memory allocations. The essence of their approach is the use of the voting function. Similar to the approach followed in this work, the authors of [GG19] transparently coalesce requests within the allocator by finding which of the threads are simultaneously calling it and using special paths for single threaded and full warp operations.

XMalloc, as proposed in [HRJ+10], is a scalable lock free dynamic memory allocator for many core systems. The authors have proposed a solution to transform the traditional atomic compare and swap based lock free algorithms to scale well on SIMD architectures. They have also proposed a hierarchical cache like buffer system to save on the average latency of accesses to non-scalable or really slow resources like the main memory in a multi core machine. They have shown XMalloc to scale well with the number of processors and also the number of vector units in every SIMD processor. They have shown their allocator to achieve 211 times better performance in terms pf speedup as compared to the normal lock-free solutions which are not truly scalable. The authors have tried to aggregate the SIMD parallel memory allocation requests into just one request. Doing this, they claim that their algorithm scales with the number of vector units in each of the processors. It is actually based on lock-free FIFO queues that hold chunks and bins of already defined sizes. The chunks are allocated from blocks of contiguous memory which has the possibility of being subdivided into any arbitrary size. Here, operations over memory blocks requires the use of locking. The current work discussed in this paper [GG19] is inspired form XAlloc in the sense that the authors have used a coarse-grained allocator for larger allocations and a fine-grained allocator for smaller allocations.

HAlloc, a high throughput dynamic memory allocator for GPGPU architectures as proposed in [Adi]. It is one of the fastest GPU memory allocator available. It defines a statically sized memory pool. This pool is divided into chunks at the time of initialization. It relies on the CUDA dynamic memory allocator for the handling of large memory allocations. It keeps one active bin for each allocation size. IT replaces the old bin with a new active bin as and when the old bin reaches a threshold which is configurable. Similar to HAlloc, the authors in [GG19] keep per-size bins in their allocator. However, they use a linked list to track all active bins, thereby avoiding the costly operations involving the replacement of the active bins.

SFMalloc, as proposed in [SKL11] is a lock-free and mostly synchronization-free dynamic memory allocator for multicore architectures. This is a special memory allocator in the sense that it does not employ any synchronization methods during multi-threaded executions. For uncommon cases, it only employs certain lock-free synchronization mechanisms. In this implementation, each of the threads is the owner of a private heap memory structure and the individual memory allocation requests are handled on this heap. Due to this, the dependency of the thread on shared memory pools is removed and there exixts no need for synchronization mechanisms. The thread allocates and deallocates the memory itself. Therefore, there is no need for inter-thread communication. For cases when a particular thread allocates memory and another thread is involved in freeing it, a lock-free stack is used to automatically add this memory back to the heap of the owner thread, to basically prevent memory blowup. This allocator also exploits various memory caching mechanisms to decrease the latency involved in management of the memory. The memory blocks which are freed are hierarchically cached in the heap of each thread and are used on memory allocation in the future. However, this allocator is not directly targeted at GPUs, but it provides an interesting orthogonal view to the issue of dynamic memory allocation in multi-threaded architectures.

The authors in [MIS+18] have proposed a non-blocking buddy system for scalable memory allocation on multi-core systems. In their work, the authors have proposed the design of a non-blocking or a lock-free allocator which implements the buddy system specifications where the simultaneous memory allocations and deallocations are handled by solely relying on Read-Modify-Write instructions executed alongside the critical path of the (de)allocation operations rather than being handled by spin-locks. The authors exploit these instructions to find if simultaneous requests have faced a collision on the same part of the allocator metadata.

In [Lee14], the authors design a novel signal model with a signal queue to handle the thread-level interaction. Based on this signal model, the authors propose to simultaneously involve the concept of buddy memory allocation to develop a non-blocking parallel buddy system.

DynaSOAr [SM18] is a parallel memory allocator for object oriented programming on GPUs with efficient memory access. It is a CUDA framework for SMMO applications. This allocator controls both the memory allocation as well as memory access. To improve the usage of allocated memory, it employs a Structure of Arrays (SoA) data layout. It is capable in achieving low memory fragmentation. This design heavily employs atomic operations. It focuses on overall application performance rather than the memory allocation and deallocation performance.

The authors in [VH15] have performed a survey of the available dynamic memory allocators for GPU architectures and provide their recommendations based on their observations. Also, they propose new allocators focusing on minimizing the number of registers used by the allocator code. They propose the use of AtomicMalloc, memory allocator proposed by them, if deallocation of memory is not needed for a specific application. Further, the authors suggest the use of Scatter-Alloc if the allocation requests made by the applications have same or similar sizes. Furthermore, they propose to employ the FDGMallocif individual threads execute many successive memory allocation requests. According to the authors, HAlloc should be used when the number of threads involved in allocating the memory is enormous. However, if the properties specific to the allocation are unknown, the authors suggest the use of CMalloc, which is also proposed by them.

5 Proposed Solution

The paper targets the problems specified in Section 3 by targeting the problems at their individual level to solve the ultimate problem of dynamic memory allocation. To enable efficient allocation of resources, the paper proposes a novel synchronization primitive, which they refer to as *bulk semaphores*. This primitive in a way generalizes the counting semaphores. Further, the authors in [GG19] propose an extension to the Read-Copy-Update mechanism for the management of linked-lists of memory blocks to allow the threads to delegate the commit operations to other thread which has already been waiting. This prevents many threads from waiting indefinitely. Furthermore, the authors propose to allow multiple concurrently running threads to access the critical section simultaneously by proposing collective synchronization primitives. Ultimately, the work targets to contribute by proposing a novel high throughput GPU memory allocator based on the previous proposals to achieve higher scalability.

5.1 Bulk Semaphores

The first proposal is to utilize **Two-stage Resource Management**. In the case of counting semaphores, as discussed in Section 2, two-stage resource allocation is used. During the first

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
                $Sem.E \leftarrow Sem.E + B - N$
                return $-1$
            else if $Sem.C \geq I$ then
                $Sem.C \leftarrow Sem.C - N$
                return 0
            else
                $Sem.R \leftarrow Sem.R + N$
            end if
        end atomic

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

        atomic
            $Sem.R \leftarrow Sem.R - N$
        end atomic
    end while
end procedure
```

Algorithm 2 SIGNAL operation on a batch semaphore

```
procedure SIGNAL($Sem$, $N$, $B$)
    atomic
        $Sem.C \leftarrow Sem.C + N$
        $Sem.E \leftarrow Sem.E - B$
    end atomic
end procedure
```

stage, the wait(N) operation allocates a given number of resources. During the second stage, a look-up and update operation is performed over a data structure to locate the resources which had been allocated. However, this two-stage resource management limits scalability, in terms of the number of threads. For instance, if a thread is allocating some new resources, all remaining threads which are trying to access the resources are blocked, even in the case where adequate resources are still present. this will lead to a sharp increase in the number of waiting threads if the number of threads is very large. To solve this issue, bulk semaphores have been proposed, which are efficient in implementing scalable two-stage resource management.

As opposed to the counting semaphores, the batch semaphores have three counters: ‘C’, which is the semaphore value (number of resource units already available), ‘E’, which is the expected counter (number of resource units expected to be available) and ‘R’, which is the reservation counter (number of units reserved by the waiting threads). The expected availability her is the value after all the expected units, E, are added to the semaphore and the reserved units have been removed. This reflects whether or not the semaphore can eventually handle the resource request coming from any thread, pertaining to the value N. Another input parameter, along with N as in the counting semaphores, ‘B’, is required in case of bulk semaphores. B is the batch size, which refers to the total resource units a thread can allocate in the case of requirement when the value C is not sufficient. This value can be added to E if the requested units is less than C. Otherwise, if C >= N, the value of C can be simply decremented by N. The wait and signal operations are summarized in Algorithms 1 and 2 respectively.

With these semantics, individual threads block on the wait operation when there are sufficient expected resource units for the supply. However, if more of these resource units are required, bulk semaphores allow many concurrently running threads release the required number of resources in batches. This solution suffices to solve the problem of false resource starvation, as mentioned in Section 3.

5.2 GPU Memory Allocator

The proposed memory allocator is based on the standard interfaces like malloc and free. However, depending upon the allocation size, the malloc request is either forwarded to *TBuddy*, a course grained memory allocator or *UAlloc*, a fine grained memory allocator.The free call is forwarded to either TBuddy or UAlloc, if the memory addressing is page aligned or not, respectively, because TBuddy guarantees page aligned memory as opposed to UAlloc, which always guarantees non page aligned memory.

This concept helps because in such a case, the need for a shared data structure to track the ownership of memory allocation is eliminated.

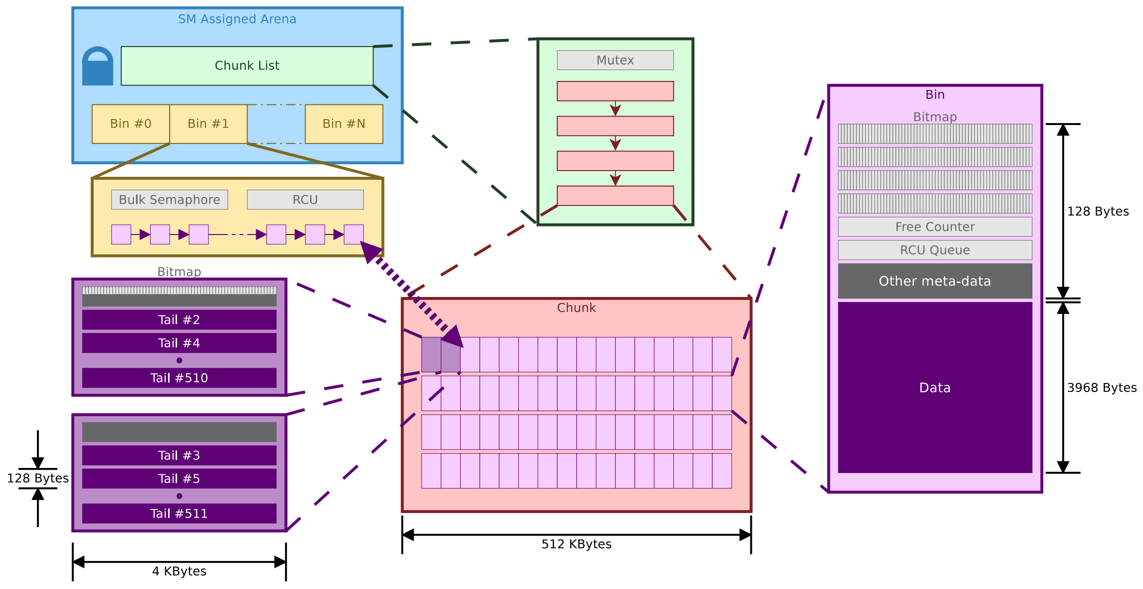


Figure 2: UAlloc: Fine-Grained Allocator

5.2.1 Tree Buddy Allocator

The memory allocations done by the OS have the ability to sustain high memory fragmentation due to the reason that the virtual address space is extremely high as compared to the physical memory. However, the memory allocator for GPUs are only allowed to allocate the requested memory from a pre-allocated pool of memory allotted by the CPU. Therefore, coarse grain memory allocations are done using a buddy system [PN77] so as to offer a decent trade-off between performance and memory fragmentation.

In a buddy system of memory allocation, a block of memory is divided into equally sized partitions until the request or demand is satisfied. It supports easy and efficient splitting and coalescing of memory blocks.This system has limited external fragmentation and allows merging and splitting of memory regions with less overhead as compared to other memory allocation techniques such as dynamic allocation.

To avoid the problem of false resource starvation, the buddy system used locks during the allocation and free operations. However, this degrades the system performance to when the number of concurrent processes competing each other as in the case of GPUs. To provide scalability to this system of memory allocation with an increase in the number of threads, two-stage resource management is used, as mentioned in Section 5. The authors call this system as the Tree Buddy Allocator.

5.2.2 UnAligned Allocator

The Unaligned Memory Allocator is inspired form the traditional memory allocation interfaces such as malloc. Under this methodology, each SM shares access to an arena, so that that cache hit rate of application is maximized.The design of UAlloc is shown in Figure 2.

The implementation of UAlloc is done using the two-stage resource management. In the first stage, the wait operation is executed in a batch semaphore. This batch semaphore is associates the the bin free list for the allocation size. Depending upon the return value of the wait operation in this scenario, the memory is allocated. If the return value is zero, a new bin is allocated from any chunk in the arena's available chunk list. However, if this list is empty, TBuddy is called.

In the second stage, the bin free list is traversed until a bin free counter is decremented without hitting a negative value.

Deferred Delegation in RCU: Some situations might arise when many threads would be allocating or deallocating memory concurrently. In such cases, The list traversal has to be protected within a critical section. However, there might be only a few threads which actually update the list, while the remaining majority would only be updating the bitmap of the bin. Therefore, it makes sense to use Read-Copy-Update as the basic mechanism for synchronization.

RCU improves the scalability as it allows concurrent reads along with updates. As compared to the traditional synchronization primitives, which strive to achieve mutual exclusion among the concurrently running threads (no considerations if they are updaters or reader), or the conventional reader-writer locks which permit concurrent reads but not while there is an update operation going on, RCU mechanism supports simultaneous operation of a single updater and multiple readers. Further, RCU ensures that the versions read by the readers are coherent by maintaining various versions of the objects and ensuring that they are not freed until and unless all the previous read operations within the critical sections are completed.

However, most of the RCU implementations rely on individual thread-based variables and the RCU operations require to iterate over all of them. RCU implementations at the GPU level will be problematic because of the high number of threads involved. Large overheads will be incurred in such a case. Therefore, the authors in [Mck08] proposed Sleepable Read-Copy-Update (SCRU), which allows reader threads to sleep or block in RCU reading-side critical sections. This provides an implementation which is suitable for the GPU. SRCU has an epoch counter and two reader counters, which track the numberof readers present in to consecutive epochs. The threads which are reading update the reader counterof the epoch as soon as it enters the RCU reading-side critical section. As it leaves this section, it decrements the value of the counter. The RCU mutex is acquired by the RCU barrier, which then increments the epoch counter, waits for the RCU counter for the previous epoch to become zero and then finally releases the RCU mutex.

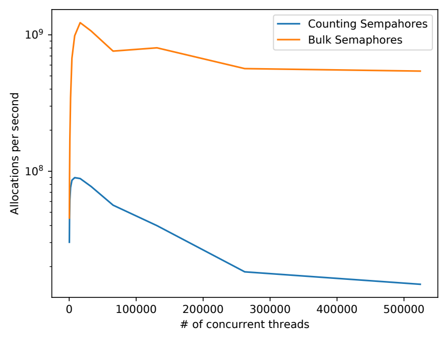
A point to be noted here is that the barriers in SRCU wait for all the threads that enter the RCU reading-side critical section before the value of the epoch counter is increased, irrespective of doing it before or after the RCU barrier being issued. However, this, as noticed, can have major impact on the performance. To solve this issue, we can exploit the property of the SRCU according to which it if an RCU barrier is issued at the time when another RCU barrier is waiting to increase the value of the epoch counter, both the barriers can be cleared at the same time. This property can be used to decrease the time during which an SM is occupied for waiting for RCU barriers to by defining conditional RCU barriers. Such a barrier returns as soon as another RCU barrier is waiting to increase the value of the epoch counter. Therefore, using this construct, it is possible that the execution of the RCU callbacks is delegated to the thread which has been waiting at the RCU barrier, thereby hastening the release of the hardware resources occupied already.

Collective Synchronization Primitives: There might be situations where several threads in a block of threads might need to allocate or de allocate a chunk off memory concurrently. To solve this issue, the authors have proposed the use of collective mutex, which allows a group of threads to collectively lock or unlock the mutex, to gain or lose access to a particular critical section. There can be two operations in this regard: collective acquire and collective release. During the first operation, one of the selected thread acquires the mutex or any other synchronization object. During the second operation, all participant threads are required to arrive at a barrier and thereafter a chosen thread releases the mutex or the synchronization object involved.

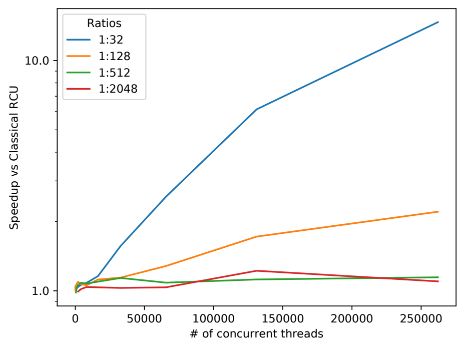
6 Evaluation

The authors in [GG19] have evaluated their proposed synchronization primitives and memory allocator using the CUDA Toolkit (v 9.2) and an NVIDIA TITAN-V GPU. Figure 3a shows the comparison between the upper limit of allocation using counting semaphores vs bulk semaphores. Here, we can see that the bulk semaphores do better than the counting semaphores due to concurrent batch allocations. As observed from the graph, the decreasing trend of allocation throughput with the increase in the number of concurrent threads because updates to the reserved value ‘R’ saturate the atomic throughput faster than updates to the expected counter ‘E’.

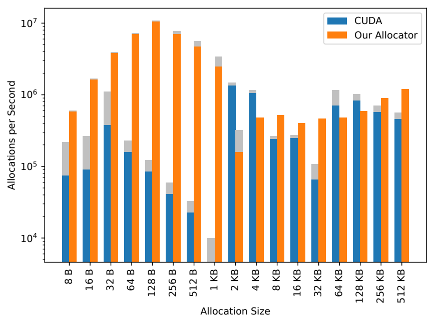
As far as the RCU delegation mechanism is concerned, it is observed from Figure 3b that decent



(a)



(b)



(c)

Figure 3: (a) Counting Semaphores vs Bulk Semaphores (b) Classical RCU vs Delegation based RCU (c) CUDA's Allocator vs Proposed Allocator

speed-up is achieved as compared to the conventional RCU mechanism for most writer:reader thread ratios. It is noticed that the performance is better when more number of threads are employed. The performance improvement in the case of delegation is not observed in the case of smaller thread counts because the possibility of two thread blocks waiting for the same set of reader-threads is extremely less.

Finally, Figure 3c shows the performance of the proposed memory allocator for different allocation sizes, as compared to the CUDA memory allocator which is based on traditional interfaces like malloc and free.

7 Discussion

The authors have specifically targeted a non-generic GPU platform, based on NVIDIA's Volta architecture [NVI17], which was proposed in 2017. The techniques are based on the fact that this architecture supports independent thread scheduling, guaranteeing forward progress of the individual threads, regardless of the control flow. Focusing a generic problem, they should have targeted the generic GPU architectures.

Also, they use techniques like buddy memory allocation. This technique specifically is known for internal memory fragmentation, though not for external memory fragmentation. Due to this, memory is wasted because the requested memory is a bit more than the available smaller memory block, but a lot smaller than the large memory block. The auhors should have used techniques such as slab allocation [B+94] to solve such issues by providing more fine grained allocation.

Further, in the introduction section, the authors themselves mention that the target domain of their work is limited to a very few applications which actually use dynamic memory allocation.

The authors did not compare their allocator results which the next in line allocator, HAlloc

[Adi], even though they mention that HAlloc outperformed their allocator when used on GPUs based on the Kepler architecture instead of the Volta architecture. This once again reflects the non-generic applicability of their work.

The authors propose a lot of ideas, but fail to synchronize the ideas and propose an ultimate solution. Also, they have provided little or no information on the actual implementation of their ideas.

However, the authors focus on a very important issue, which certainly demands more attention in todays world. The issue of memory handling while using enormous concurrency in terms of number of threads is an important issue. Their individual proposals like new synchronization primitives and bulk semaphores are also good contributions.

8 Conclusion

The paper targets a very important field in case of concurrent programming. People have been looking at GPUs as a solution to improvise on concurrent programming. However, various issues regarding the scalability of synchronization primitives with the increasing number of threads have been faced. Most of these issues reflect on higher levels, such as memory allocation. The authors have proposed a novel dynamic memory allocator based on various improvisations such as two-stage resource allocator, bulk semaphores, collective synchronization primitives and delegation of deferred reclamation to threads which are already stalled. The allocator achieves 16.56 times better allocation rates as compared to the CUDA memory allocator. However, there are some shortcomings which do not seem to have been acknowledged in this work, one of them being a generalized memory allocator based on the proposed improvisations. The proposed allocator is suitable for NVIDIA's Volta architecture specifically.

References

[Adi] Andrew V Adinetz. Halloc: a high-throughput dynamic memory allocator for gpgpu architectures. 4, 7

[B$^+$94] Jeff Bonwick et al. The slab allocator: An object-caching kernel memory allocator. In *USENIX summer*, volume 16. Boston, MA, USA, 1994. 7

[GG19] Isaac Gelado and Michael Garland. Throughput-oriented gpu memory allocation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 27–37. ACM, 2019. (document), 4, 5, 6

[HRJ$^+$10] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W. Hwu. Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 1134–1139, June 2010. 4

[Lee14] Jon-Yu Lee. A new non-blocking approach on gpu dynamical memory management. In *The 2013 International Workshop on Computational Science and Engineering*, volume 202, page 071. SISSA Medialab, 2014. 4

[Mck08] Paul Mckenney. Sleepable read-copy update. 10 2008. 5.2.2

[MIS$^+$18] R. Marotta, M. Ianni, A. Scarselli, A. Pellegrini, and F. Quaglia. A non-blocking buddy system for scalable memory allocation on multi-core machines. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 164–165, Sep. 2018. 4

[NVI17] Tesla NVIDIA. V100 gpu architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017. 7

[OTT02] Author One, Author Two, and Author Three. The title of a VERY important paper you have been given. *The publishing journal, conference, etc.*, 20(1):1–24, February 2002.

[PN77] James L Peterson and Theodore A Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977. 5.2.1

[SKKS12] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. Scatteralloc: Massively parallel dynamic memory allocation for the gpu. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10. IEEE, 2012. 4

[SKL11] S. Seo, J. Kim, and J. Lee. Sfmalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 253–263, Oct 2011. 4

[SM18] Matthias Springer and Hidehiko Masuhara. Dynasoar: A parallel memory allocator for object-oriented programming on gpus with efficient memory access, 2018. 4

[VH15] M. Vinkler and V. Havran. Register efficient dynamic memory allocator for gpus. *Comput. Graph. Forum*, 34(8):143–154, December 2015. 4

[WWWG13] Sven Widmer, Dominik Wodniok, Nicolas Weber, and Michael Goesele. Fast dynamic memory allocator for massively parallel architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 120–126, New York, NY, USA, 2013. ACM. 4