High Performance Graphics (2014)
Jonathan Ragan-Kelley and Ingo Wald (Editors)

Register Efficient Memory Allocator for GPUs

M. Vinkler[1] and V. Havran[2]

[1]Faculty of Informatics, Masaryk University, Czech Republic
[2]Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic

Abstract

We compare four existing dynamic memory allocators optimized for GPUs and show their strengths and weaknesses. In the measurements we use three generic evaluation tests proposed in the literature and add one with a real workload where dynamic memory allocation is used for building the kd-tree data structure. Following the performance analysis we propose a new dynamic memory allocator and its variants that address the limitations of the existing dynamic memory allocators. The new dynamic memory allocator uses few resources and is targeted towards large and variably sized memory allocations on massively parallel hardware architectures.

Categories and Subject Descriptors (according to ACM CCS): D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming D.3.3 [Programming Languages]: Language Constructs and Features—Dynamic storage management D.4.2 [Operating Systems]: Storage Management—Allocation/deallocation strategies

1. Introduction

The increase in capabilities and performance of GPUs allows for programming techniques known from the CPUs to be used on GPUs as well. One of such techniques is the ability to dynamically allocate memory directly from the kernel running on the GPU. This ability opens up new possibilities for implementing algorithms targeting GPUs and allows for entirely new algorithms to be ported to GPUs. While a significant amount of work has been devoted to dynamic memory allocations on the CPU, the GPU memory allocation has only recently drawn some interest. The design of a GPU dynamic memory allocator (further referred to as allocator) is a challenging task because of the massively parallel nature of GPUs. Thousands of threads may be allocating memory at the same time making the state-of-the-art CPU allocators computationally or memory inefficient.

In this paper we compare and analyze several formerly published and available algorithms for dynamic memory allocations on the GPU. Then we propose a new allocator that addresses some of the limitations found in the current solutions. In particular, we target allocators that behave well in real workloads, in our case the construction of a kd-tree data structure. In such scenarios, factors like the register usage directly influence the performance of the allocation (acquiring a pointer to a continuous chunk of memory of a given size) and deallocation (returning the chunk so that it can be reused by upcoming allocations) operations and may lead to different results than when using generic tests.

The paper is structured as follows. Section 2 summarizes the literature on dynamic memory allocations on the GPU. The existing allocators included in our comparison are described in more detail in Section 3. In Section 4 we propose our new allocator for GPUs. The evaluation tests used for the comparison are defined in Section 5. Section 6 gives the measurements of the allocators performance in the evaluation tests. Finally, Section 7 summarizes our findings and presents possibilities for future research.

2. Related Work

The design of allocators for single-core and multi-core systems is a well researched area. To achieve low latency and good caching behavior the state-of-the-art CPU allocators maintain one private heap per thread [SKL11, LC12]. Such technique does not scale well to many-core architectures with thousands of threads running concurrently because it introduces too high memory overhead [SKKS12].

In recent years several researchers have shown interest in the area of dynamic memory allocation on the GPU. The built-in allocator distributed with the CUDA framework [NBGS08]

DOI: 10.2312/hpg.20141090

was introduced in version 3.2 of the SDK [NVI10]. We refer to this allocator as `CudaMalloc`.

Huang et al. [HRJ*10] presented a two-level allocation scheme for the many-core architectures called `XMalloc`. On the first level, allocations of superblocks, serving subsequent allocations, are handled by updating a doubly linked list that defines the usage of the memory pool (a continuous block of memory). On the second level, individual chunks of memory are allocated from these superblocks. Deallocated chunks of memory are cached for faster reuse. To accelerate concurrent allocations by threads from the same SIMD unit (a warp) the individual allocations are coalesced into a single allocation and performed by a single thread of the warp.

Because of the addition of hardware caches in the newer generation of GPUs, an updated version of the `XMalloc` allocator was presented by the same authors [HRJ*13]. In this version the allocation of superblocks in the doubly linked list is replaced with `CudaMalloc` allocator introduced in the newer version of CUDA. The caches for the fixed-size lists of deallocated items were also changed.

The dynamic allocation of GPU memory in CUDA was further researched by Steinberger et al. [SKKS12] who call their allocator `ScatterAlloc`. They ported some of the current CPU allocators to the GPU and showed that those are not efficient in the context of massively parallel processors. Based on the findings they set the design goals for a GPU allocator. They compare `ScatterAlloc` against `CudaMalloc` and `XMalloc` showing that hashing significantly decreases the allocation times and thus `ScatterAlloc` outperforms the previous approaches.

Another CUDA allocator `FDGMalloc` was proposed by Widmer et al. [WWWG13]. The authors claim that their allocator is faster than `ScatterAlloc` by a factor of 10 to 1000. We discuss `ScatterAlloc` and `FDGMalloc` in more depth in the next section as they are among the fastest allocators. A similar idea to Widmer et al. was presented by Grimmer et al. [GKR13]. Their method uses some CPU management and allows for deallocating of individual memory allocations. The authors conclude that their allocator is slower than `ScatterAlloc`.

Recently, a memory allocator has also been developed for OpenCL by Spliet et al. [SHGV14]. In OpenCL the hardware specifics are less exposed to the programmer than in CUDA making the implementation more challenging. Their allocator is faster than `CudaMalloc` on NVIDIA devices, but significantly slower than `ScatterAlloc` through an indirect comparison. Very recently Steinberger et al. [SKK*14] have proposed a list based allocator for allocating geometry buffers that is based on the buddy allocator scheme of Knowlton [Kno65].

3. Existing Allocators

We use four existing allocators with available source codes or binaries for our comparison. We do not use `XMalloc` as it was shown [SKKS12] to be slower than `ScatterAlloc`. From the results given in their corresponding papers these four allocators should include the fastest allocators for GPUs. In this section we describe in more detail how they operate and highlight their strengths and weaknesses.

3.1. AtomicMalloc

The simplest allocator can be implemented by using a single atomic instruction (see Algorithm 1). This allocator was hinted in several publications (e.g. Tzeng et al. [TPO10]), but to the best of our knowledge it was never formally described and tested for performance. The atomic addition (*atomicAdd(L,N)*) function takes two arguments: a memory address A and an integer N. It atomically performs $A \leftarrow A+N$ and returns the value previously stored at address A.

Algorithm 1: `AtomicMalloc` allocation.

```
1 mallocAtomicMalloc(size) begin
2     offset ← atomicAdd(memOffset, size);
3     return mem + offset;
```

An atomic addition to a single global variable *memOffset* gives each allocating thread a unique *offset* to the beginning of a memory chunk of size *size*. The allocated pointer is computed by adding the pointer to the start of the memory pool *mem* and the returned *offset*. This kind of memory allocation consisting of a single instruction should be the fastest possible. However, due to a single point of conflict caused by the hardware executing this instruction atomically, threads are serialized in their execution, slowing down the allocator. Moreover, memory cannot be deallocated because the returned offset only increases. Nevertheless, some tasks e.g. allocation of a new node of a data structure do not require deallocation of the memory.

3.2. CudaMalloc

Another allocator we use in our comparison is the one built-in into the CUDA framework [NBGS08]. This allocator uses an unpublished algorithm and was reported by several papers to be rather slow [SKKS12, HRJ*13, WWWG13].

3.3. ScatterAlloc

Further, we use the allocator `ScatterAlloc` of Steinberger et al. [SKKS12] which is targeted towards many parallel allocations with roughly the same size. Their method pre-splits the memory pool into pages with regular size and groups them into blocks. During allocation hashing is used to select a page from the currently used block, which causes

the allocations to be distributed in memory and prevents conflicts of atomic operations. The page itself can then be split into a maximum of 2^{10} chunks that are individually allocated and deallocated. If memory larger than the page size is requested, the allocating thread tries to lock successive pages to serve the request. If the thread fails to acquire all the needed pages, it unlocks the ones already locked and restarts the search in another location. To lower the probability of restarting, only a single thread may try allocating a large memory chunk. This serializes all allocation requests of large memory chunks and is only efficient when a few threads are allocating large chunks of memory concurrently.

3.4. FDGMalloc

We also use the `FDGMalloc` of Widmer et al. [WWWG13]. In their allocator each warp requests large blocks (superblocks) of memory from a global memory pool using the built-in `CudaMalloc` similar to the method of Huang et al. [HRJ*13]. Each warp manages its own list of allocated superblocks without any synchronization with other warps. For the allocation inside these superblocks no header information is used, only the pointer to the unoccupied memory is updated. This limits the memory overhead of individual allocations but at the same time prevents the allocations from being deallocated separately. The allocated memory can only be deallocated all at once. If only a single allocation or allocation larger than the superblock size is requested, the allocator's behavior is determined by the `CudaMalloc` serving these requests. However, when multiple allocations smaller than the superblock size are requested by a warp these are served very quickly.

4. Proposed Allocators

All of the allocators summarized in the previous section have some limitations. `AtomicMalloc` cannot reuse memory, `CudaMalloc` is generally very slow, and `ScatterAlloc` and `FDGMalloc` are biased towards small and repetitive allocations. In this section we propose allocators that alleviate some of these limitations.

4.1. AtomicWrapMalloc (AWMalloc)

We propose a variant of `AtomicMalloc` that is capable of memory reuse. This modification allocates memory in a circular memory pool (see Algorithm 2).

If enough memory is present, the allocator works the same way as `AtomicMalloc`. On the other hand, when the new allocation does not fit into the memory pool (returned *offset* plus the requested *size* is larger than the size of the memory pool *memorySize*) a wrapped allocation from the beginning of the memory pool is attempted using *atomicCAS*. The atomic Compare-and-Swap (*atomicCAS(A,E,N)*) function takes three arguments: a memory address A, a value E expected to be stored at the location, and a value N to replace the expected value. If E is indeed stored at address A, it is atomically replaced with N. In any case the value stored at memory given by address A at the time of the atomic operation is returned. If the wrap is not successful due to some other thread modifying the *memOffset*, two cases may occur. Either the *memOffset* was already wrapped by some other thread in which case a new allocation using *atomicAdd* is attempted or the wrapped allocation is attempted again using the returned offset *newOffset*.

Algorithm 2: `AWMalloc` allocation.

```
1  mallocAWMalloc(size) begin
2      offset ← atomicAdd(memOffset, size);
3      newOffset ← offset + size;
4      while (newOffset > memorySize) do
5          newOffset ← atomicCAS(memOffset,
           newOffset, size);
6          if (newOffset = offset + size) then
7              return mem;
8          else if (newOffset + size) ≤ memorySize then
9              offset ← atomicAdd(memOffset, size);
10             newOffset ← offset + size;
11         else
12             offset ← newOffset − size;
13     return mem + offset;
```

The wrap may result into an overwrite of the previously allocated memory, breaking the correctness of the computation. However, if a large enough memory pool is used and the threads use the allocated memory for only brief periods, this allocator can provide the performance of `AtomicMalloc` with significantly decreased memory requirements.

Given the memory reuse limitations of `AtomicMalloc` and `AWMalloc` they cannot be considered as full allocators. We include their measurements in this study mostly for comparison to other allocators since their performance can be considered as a lower bound.

4.2. CircularMalloc (CMalloc)

Our second proposed allocator organizes the memory pool as a singly linked list. We have also attempted to organize the memory pool as a doubly linked list, but it proved slightly slower in our measurements.

Similar to `AWMalloc` we allocate memory from a circular memory pool. Since we design an allocator that is able to also deallocate memory, each allocated chunk of memory is prefixed with a header. This header consists of two 4 Byte words: the allocation flag and the offset of the next chunk. Allocating from a memory pool consisting of a single free

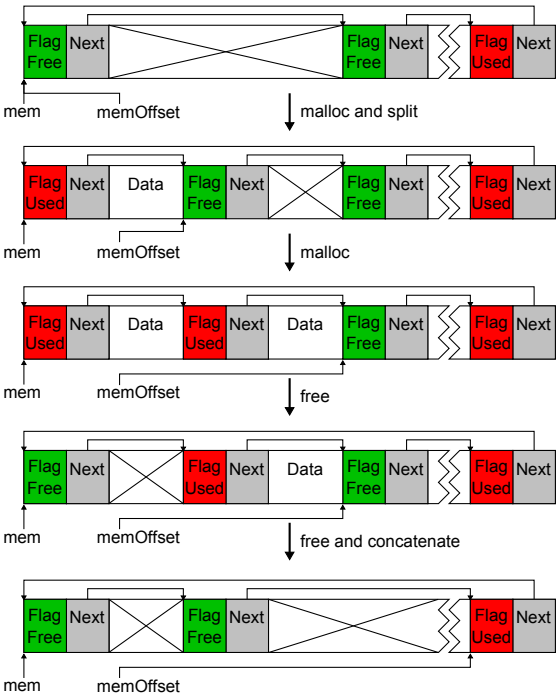


Figure 1: `CMalloc` allocation and deallocation. The header for each allocation consists of the flag and the next pointer. The memory pool is pre-split into arbitrarily sized chunks and the end of the memory pool is marked with a header that stores a used flag and a pointer to the beginning of the memory pool, making the memory pool circular. Depending on the size of the allocation request and of the current chunk a new chunk may be created during allocation. During deallocation a chunk may be merged with the next free chunk.

chunk would require serialization of the threads locking the chunk's flag as in the `AWMalloc` but with a much slower time for a single allocation. The example of allocation and deallocation using `CMalloc` is shown in Figure 1.

We propose to pre-split the memory pool into many free chunks as in `ScatterAlloc`. However, in our allocator the chunk sizes need not be uniform and can be optimized for the algorithm that uses the allocator. We pre-split the memory pool into chunks of size $C(i)$:

$$C(i) = R/2^{\lfloor log_2(i) \rfloor}, \quad (1)$$

where $C(1)$ is the size of the first, largest chunk. This formula for the chunk sizes creates a structure similar to a binary heap, where R is the size of the root node (see Figure 2). The R is computed so that the largest binary heap fits into the memory pool. The memory not used by the heap forms the last chunk. In our tests such division provided in most cases the best performance while having high flexibility in the size of the allocation requests that can be served by the allocator.

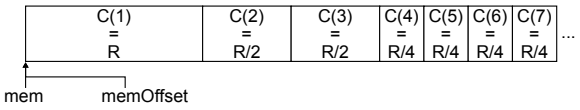


Figure 2: Memory pool pre-splitted into a binary heap.

The pseudocode for the `CMalloc` allocator is shown in Algorithm 3. The offset to the beginning of the allocated memory of size *size* is acquired by finding a first large enough free chunk starting from a chunk at the offset stored in the shared global variable *memOffset*. This chunk has to be large enough to contain both the requested *size* Bytes and the allocation header of size *HEADER_SIZE* (8 Bytes). Within this header the pointer to the next chunk is stored at position *NEXT_OFS* (4 Bytes) after the header start.

During allocation we use atomic Compare-and-Swap function on the chunk's flag in an attempt to set it as used. If the acquired chunk is larger than the requested size times the maximum allowed fragmentation *MAX_FRAG*, the chunk is split into two. After the newly created chunk header is written to the memory pool the allocating thread has to wait for the memory write to be visible to the other threads before linking the chunk. The *__threadfence* [NVI12] function from the pseudocode blocks the executing thread until the memory writes performed by this thread prior to calling the function are guaranteed to be visible to all other launched threads. Without this call the list might be in incoherent state for the other threads. Then the *memOffset* is set to the chunk following the recently allocated one. The final pointer to the allocated memory is computed by adding the pointer to the start of the memory pool *mem*, the offset of the allocated chunk *offset* and the size of the header *HEADER_SIZE*.

When the memory is deallocated, the chunk can be possibly merged with the following memory chunk. If the following chunk is free and can be set as used (so that no other thread may use it), the next pointer of the chunk being deallocated is set to the next pointer of the following chunk. Again we have to wait for the memory write to be visible to other threads before setting the chunk's flag as free. For some sequences of allocations and deallocations merging only with the following chunk may lead to a very fragmented memory pool. In such a case the doubly linked list is a better option or the buddy allocator scheme of Knowlton [Kno65] can be used.

Although list-based allocators have been previously tested [HRJ*10, SKKS12] our approach uses the GPU resources efficiently leading to high performance. Compared to the allocator of Huang et al. [HRJ*10] our allocator uses a simpler implementation of a linked list. In particular, we utilize only one level of allocations and no custom caches are used for deallocated items. This is possible thanks to the advances in the graphics hardware, mainly the addition of hardware cache hierarchies and improved atomic instructions.

Algorithm 3: `CMalloc` allocation and deallocation.

```
1  mallocCMalloc(size) begin
2      offset ← memOffset;
3      size ← size + HEADER_SIZE;
4      while (true) do
5          lock ← atomicCAS(mem[offset], Free, Used);
6          next ← mem[offset + NEXT_OFS];
7          csize ← next − offset;        // Chunk size
8          if (lock = Free) then
9              if (size ≤ csize) then
10                 break;
11             mem[offset] ← Free;
12         offset ← next;
13     newNext ← next;
14     if (size + HEADER_SIZE ≤ csize and
15         size·MAX_FRAG ≤ csize) then
16         newNext ← offset + size;
17         mem[newNext] ← Free;
18         mem[newNext + NEXT_OFS] ← next;
19         __threadfence();
20         mem[offset + NEXT_OFS] ← newNext;
21     memOffset ← newNext;
22     return mem + offset + HEADER_SIZE;

23 freeCMalloc(ptr) begin
       // Access the header before the
       // data pointed to by ptr
24     next ← ptr[−HEADER_SIZE + NEXT_OFS];
25     if (atomicCAS(mem[next], Free, Used) = Free)
       then
26         next ← mem[next + NEXT_OFS];
27         memOffset ← next;
28         ptr[−HEADER_SIZE + NEXT_OFS] ← next;
29         __threadfence();
30     ptr[−HEADER_SIZE] ← Free;
```

4.3. Circular Fused Malloc (CFMalloc)

Given the flag for each chunk holds just the free / used information it can be contained in a single bit of data. In case fewer than 2^{31} words need to be allocable the flag can be fused with the offset of the next chunk (a 32-bit value). This allows only one word of memory to be read or written to during allocation and deallocation leading to a simpler code.

4.4. Circular Multi Malloc (CMMalloc) & Circular Fused Multi Malloc (CFMMalloc)

The two previous allocators can be further extended to achieve higher performance at the cost of increased memory fragmentation. The single offset into the memory pool *memOffset* can be replaced with an array of offsets (one for each streaming multiprocessor) pointing initially to different chunks. This technique corresponds to the hashing used in `ScatterAlloc` and decreases the number of conflicts of atomic operations during the allocation and deallocation.

We also change the sizes of the pre-split chunks in the memory pool. Each of the offsets in the array initially points to a first chunk of a heap like structure created according to equation 1. The individual heaps use the size of the root chunks $R' = R/\#SM$, where $\#SM$ is the number of streaming multiprocessors on the GPU. These smaller heaps are linked together in a singly linked list, keeping the same structure of the memory pool as in the two previous allocators.

5. Evaluation Tests

To test the allocators we have implemented three generic evaluation tests introduced in the previous papers on dynamic memory allocation on GPUs. In addition to these we have developed a real workload for the memory allocation.

[AD] Alloc Dealloc. This simple test kernel allocates memory for each warp and then immediately deallocates the memory [HRJ*10, p. 5] .

[ACD] Alloc Cycle Dealloc. This test kernel [WWWG13, p. 5] is an extension of the previous one. Inside a single kernel multiple iterations of allocation are requested followed by deallocation of all of the allocated memory. Multiple allocations increase the pressure on the allocator and show the allocators ability to optimize these subsequent allocations.

[P] Probability. This test [SKKS12, p. 7] is again a variant of the first test. Each kernel launches a memory allocation with a probability $pAlloc = 0.75$ if there is no currently allocated memory, and deallocates the memory with probability $pFree = 0.75$ if there is an allocated memory. The same kernel is called multiple times so that a more complex mix of allocations and deallocations emerges.

[DS] Data structure build. We build a spatial data structure (kd-tree) on the GPU. In this data structure primitives may be duplicated in both the left and right children of the split node. To solve this duplication of primitives two new chunks of memory have to be allocated to hold the primitives in the left and right child nodes. This scenario is highly challenging since a large scale of allocation sizes could be requested during the build and the order of the allocations is unknown.

We have decided to let a single thread of each warp allocate memory in all of these tests. The techniques for coalescing memory allocations inside a SIMD unit are well researched and used in both `ScatterAlloc` and `FDGMalloc` as well as in the previous literature on GPU dynamic memory allocations [HRJ*10]. The same or similar code may thus be used for the other allocators with the same influence on measured performance. The memory requested in each allocation is padded to a multiple of 16 Bytes in our tests to have the same request sizes for all allocators

since ScatterAlloc and FDGMalloc are doing so internally [SKKS12, WWWG13].

6. Results

We evaluated the allocators on a PC with Intel Core i7-2600, 16 GB of RAM and NVIDIA GTX TITAN Black running 64-bit Windows 7 and CUDA Toolkit 4.2. The compared existing allocators should represent the fastest solutions to this date. We measure the entire GPU time of a test using CUDA events [NVI12]. Each test was run five times, with the median of the test times reported. Figure 3 shows several properties of the compared allocators using the generic tests. The setting for all generic evaluation tests is given in Table 1.

Test	#Iters [-]	#Blocks [-]	Heap Size [B]	Payload [B]	Figure
[AD]	1	120	2GB	4B – 128KB	3(a)
[ACD]	10	120	2GB	4B – 128KB	4
[ACD]	10	120	128KB – 2GB	4B	3(b)
[ACD]	10 – 100	120	2GB	4B	3(c)
[P]	10	1 – 120	2GB	4B	3(d)

Table 1: The setting of our tests. The number of allocations (and deallocations) performed by all threads is *#Blocks* × 8 (warps per block) × *#Iters* except for the Probability test where the number of allocations is 1450 and the number of deallocations is 986 when launching 120 blocks.

[AD] Alloc Dealloc. First, we tested the influence of increasing the size of memory requested by the threads (see Figure 3(a)). CudaMalloc and FDGMalloc are by far the slowest allocators in this test, with FDGMalloc being slower than CudaMalloc. This comes from the fact that FDGMalloc also allocates its header data using CudaMalloc. While for most allocators the performance is nearly constant, for ScatterAlloc it is not. When the request size exceeds the page size there is a sharp decrease in performance.

[ACD] Alloc Cycle Dealloc. Since the performance of ScatterAlloc depends on the size of its pages we have evaluated this influence in Figure 4. The resulting graph shows several properties of the allocator. If the request size exceeds the page size and a special allocation path for large requests is used, the time of the evaluation test becomes prohibitively high. Using a larger page size is not always a solution, since this increases the allocation times for all request sizes. Also no available chunk may be found for very large request sizes as indicated by the missing data point for the highest page size. Moreover, for page sizes higher than 64MB the test fails completely.

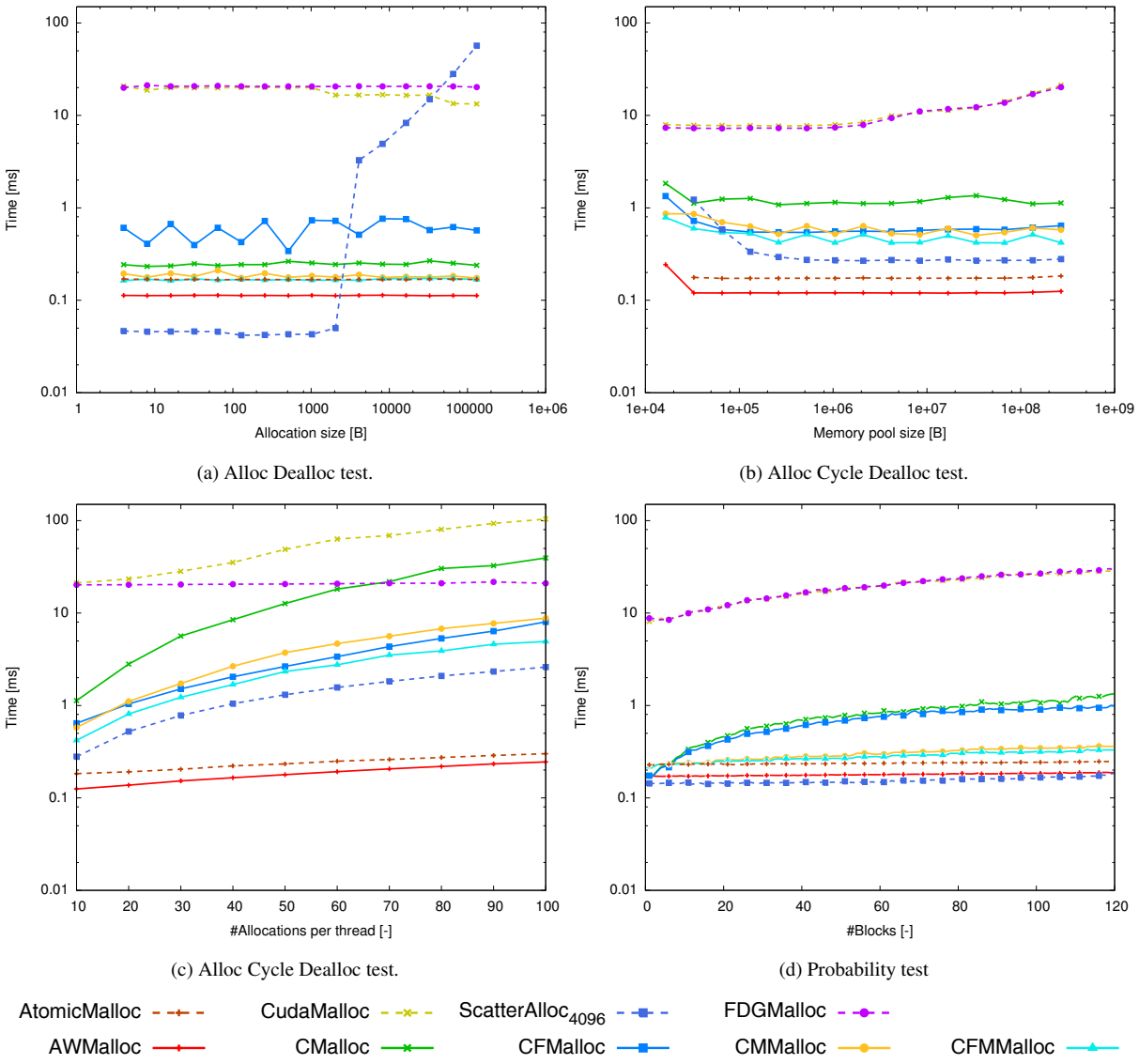
Figure 3(b) shows the dependence of the time of the Alloc Cycle Dealloc test on the size of the memory pool. The performance of CudaMalloc degrades with the increasing size of the memory pool and so does FDGMalloc which uses CudaMalloc. FDGMalloc is slightly faster than CudaMalloc in this test since the first allocation of a superblock is reused in the nine subsequent allocations. ScatterAlloc is slower for smaller sizes of the memory pool than for larger ones as conflicts of atomic instructions are more likely to occur due to imperfect hashing. The performance of the other allocators stays nearly the same regardless of the size of the memory pool.

The graph in Figure 3(c) shows the influence of increasing the number of successive allocations in the Alloc Cycle Dealloc test. While the test time for most of the allocators increases linearly, the time for FDGMalloc stays almost constant. This is caused by successive allocations being handled separately by each warp in its own superblock without any synchronization with other warps. More than 70 iterations are needed for FDGMalloc to surpass the performance of the allocators other than CudaMalloc. However, FDGMalloc may use any of the faster allocators for the allocation of superblocks, while keeping the constant performance in iterative allocations.

[P] Probability. Last, we tested the behavior under an increasing number of threads allocating memory in the Probability test (see Figure 3(d)). We launched an increasing number of thread blocks containing 256 threads each. Since only the first thread of a warp allocates memory (as discussed in the previous section) 8 allocating threads are added with each added block. All allocators show nearly linear scaling in this test with AtomicMalloc, AWMalloc and ScatterAlloc having an almost flat slope.

The performance of ScatterAlloc can approach the one of AtomicMalloc and sometimes even slightly surpass it because there are fewer conflicts of atomic operations due to hashing. For the same reason AWMalloc is faster than AtomicMalloc in all of the generic tests; the extra code causes variations in the time allocating threads access the shared variable, alleviating the serialization. Limiting the number of conflicts of atomic operations through using multiple offsets also makes CMMalloc and CFMMalloc faster than CMalloc and CFMalloc. Comparing our fastest allocator CFMMalloc to ScatterAlloc in these generic tests we observe that our allocator is 1.50× to 1.97× slower as shown in Figures 3(c) and 3(d). This is likely due to the conflicts of atomic operations.

[DS] Data structure build. The generic tests where only the allocation and deallocation operations are performed and all threads allocate memory at the same time may not represent the real workloads well. For this reason we also compare the kd-tree build times when using different allocators (see Table 2). The size of the memory pool is set to 200× the size of the root node, which is sufficient to run the build even with AtomicMalloc allocator. The FDGMalloc is excluded from this test because it can only deallocate all of the memory at the end of the computation. Moreover, the memory



(a) Alloc Dealloc test.

(b) Alloc Cycle Dealloc test.

(c) Alloc Cycle Dealloc test.

(d) Probability test

Figure 3: Graphs showing the properties of the allocators in various benchmarks. (a) Alloc Dealloc test — increasing the size of memory requested in each allocation, (b) Alloc Cycle Dealloc test — increasing the size of the memory pool, (c) Alloc Cycle Dealloc — increasing the number of allocations per thread before the memory is deallocated, (d) Probability test — increasing the number of allocating threads (8× the number of blocks).

is tied to a particular warp requiring synchronization with other warps before the memory can be deallocated. Although the memory cannot be deallocated for `AtomicMalloc` and `AWMalloc` as well, we chose to add results for them since their performance can be considered as a lower bound on the kd-tree build time.

From Table 2 we can observe that `ScatterAlloc`, which proved to be almost as fast as `AtomicMalloc` in the generic tests, is significantly slower in this real workload. There are two reasons for this. First, the allocation sizes vary by several orders of magnitude in this test, which is problematic for the fixed page size of `ScatterAlloc`. Second, `ScatterAlloc` consumes significantly more registers than `AtomicMalloc` and even `CMalloc`. When not enough registers are present for the entire kernel the compiler optimizes the code to use fewer registers, slowing down the computation. `ScatterAlloc` is by far the most demanding allocator, requesting almost the entire register file

	Scene										
	Hand Stand	Sponza	Sibenik	Fairy Forest	Crytek Sponza	Conference	Armadillo	Dragon	Happy Buddha	Blade	Sodahall
N_{tris} [−]	20K	76K	80K	174K	262K	283K	307K	871K	1,087K	1,765K	2,169K
N_{ref} [−]	6.12	6.79	4.70	5.42	6.30	5.96	6.63	7.95	8.84	8.35	5.28
N_{alloc} [−]	65K	141K	101K	240K	509K	251K	943K	3,740K	5,635K	7,286K	2,981K
$\bar{X}_{\|alloc\|}$ [−]	45.8	81.8	100.8	101.8	79.8	167.8	51.9	44.5	40.2	45.7	113.2
$\sigma_{\|alloc\|}$ [−]	335.6	594.4	734.7	791.1	638.1	989.5	458.1	388.3	362.1	394.0	721.0

Allocator	#Registers											
		T_{build} [ms]										
`AtomicMalloc*`	4	7.29	18.38	14.6	26.4	43.2	31.5	65.2	378.9	918.3	2175.1	325.5
		Slowdown [−]										
`CudaMalloc`	6	5.42	2.99	2.28	2.73	3.82	2.85	5.19	20.95	19.05	12.13	3.38
`ScatterAlloc`$_{4096}$	38	1.66	1.74	1.39	1.56	2.01	2.67	2.15	1.78	2.08	1.05	5.08
`ScatterAlloc`$_{8192}$	38	1.99	1.65	1.46	1.52	1.84	1.51	2.52	7.63	1.73	1.07	2.82
`ScatterAlloc`$_{16384}$	38	1.78	1.70	1.55	1.79	2.47	1.60	3.52	6.90	6.87	4.99	1.87
`FDGMalloc*`	26	-	-	-	-	-	-	-	-	-	-	-
`AWMalloc*`	6	1.00	0.88	0.94	0.87	0.96	0.83	1.00	0.90	1.07	0.96	0.95
`CMalloc`	16	1.48	**1.04**	1.18	1.17	1.14	**1.08**	1.16	1.27	**1.20**	0.99	**1.14**
`CFMalloc`	10	1.31	1.20	**1.10**	**1.06**	**1.08**	1.17	**1.12**	**1.17**	1.26	**0.93**	1.17
`CMMalloc`	16	1.37	1.10	1.16	1.12	1.20	1.09	1.23	1.27	2.00	1.03	1.17
`CFMMalloc`	14	**1.30**	1.08	1.11	1.12	1.25	1.12	1.19	1.24	1.34	1.03	1.20

Table 2: The allocation properties and build times of kd-trees when using the compared allocators. N_{tris} is the number of scene triangles, N_{ref} is the number of triangle references from leaves, N_{alloc} is the number of allocation (and deallocation) requests during the build, $\bar{X}_{|alloc|}$ is the mean of the allocation sizes and $\sigma_{|alloc|}$ is the standard deviation of the allocation sizes. For each allocator the number of registers used in a kernel consisting of a single allocation and deallocation is reported (#Registers). For `AtomicMalloc` the build times of kd-trees are given in milliseconds and this allocator is taken as the reference. For the other allocators the ratio of their build times and the build time of the reference is reported. The fastest allocator for each scene is type-setted in boldface. The allocators marked with an asterix are not full allocators and are not considered as being the fastest.

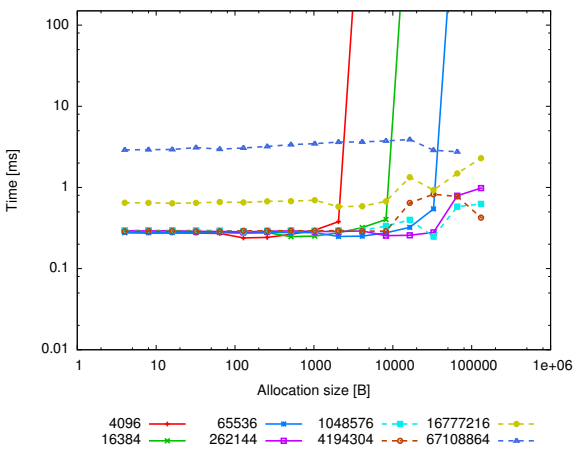


Figure 4: Dependence of `ScatterAlloc` on the page size (the individual curves) in the Alloc Cycle Dealloc test. The exact setting of the test is in Table 1.

when targeting full occupancy (maximum number of threads capable of running simultaneously at the streaming multiprocessor). With the kd-tree build also being register hungry this causes `ScatterAlloc` to be slower than the other allocators except `CudaMalloc`. However, this influence on the allocators performance is hard to evaluate precisely since artificially lowering the register pressure accelerates both the allocator and the kd-tree build algorithm.

Contrary to the generic tests `CMMalloc` and `CFMMalloc` allocators are often slower than `CMalloc` and `CFMalloc` when used in the kd-tree build. In this test, the warps are not requesting memory allocations at the same time during the build cancelling the usefulness of multiple offsets into the memory pool.

We have also tested the allocators on a different GPU: the NVIDIA GeForce GTX 680. This GPU has 8 SMs (streaming multiprocessors), while the NVIDIA GTX TITAN Black features 15 SMs. This reduction in the number of processors results into fewer conflicts of atomic operations on a shared

variable. In the generic test the GTX 680 was almost twice as fast for all of the allocators except `ScatterAlloc` which is less prone to these conflicts.

7. Conclusions and Future Work

In this paper we have propose a new dynamic memory allocator designed for GPUs and its variants and compared them to four existing allocators using three generic and one real workload evaluation tests.

We can provide these recommendations for applications with allocation properties similar to our generic tests: `AtomicMalloc` or `AWMalloc` should be used if deallocation of memory is not needed. `ScatterAlloc` should be used if the allocation requests have similar sizes, the memory pool is large and enough registers are present for the kernel. `FDGMalloc` should be used if each thread performs a large number of successive allocation requests and `CMalloc` or its variants should be used if the allocation properties are unknown.

We showed that the limitations of the existing dynamic memory allocators, which may not manifest in the generic tests, cause significant slowdown in the real workloads. For applications with large variability in allocation sizes or high register pressure our proposed simple dynamic memory allocator `CMalloc` and its variants are capable of outperforming the state-of-the-art dynamic memory allocators. For the memory allocation/deallocation pattern of the parallel kd-tree building algorithm the speedup computed from the whole running time when using `CFMalloc` compared to `ScatterAlloc` with tuned page size is between 1.13× and 4.33×. In addition our `CMalloc` does not need any parameters to be set.

To ease the use of the new allocators we provide their source codes in the form of a library that can be accessed at `http://decibel.fi.muni.cz/~xvinkl/CMalloc/`. In future work we would like to investigate the pre-splitting strategies of the memory pool in `CMalloc` allocator for various real workloads. The more involved hybrid allocation strategies could also be investigated.

Acknowledgement

We would like to thank the contributors of the scenes used for measurements in this paper. Marko Dabrovic (http://www.rna.hr) for the Sponza and Sibenik scene, DAZ3D (www.daz3d.com) for Fairy Forest, Frank Meinl at Crytek for the Crytek Sponza model, Anat Grynberg and Greg Ward for the Conference scene, Carlo Séquin for Sodahall model and Stanford 3D Scanning Repository for the other models.

This research was partially supported by the Czech Science Foundation under project No. P202/12/2413 (Opalis) and the Grant Agency of the Czech Technical University in Prague, grant No. SGS13/214/OHK3/3T/13.

References

[GKR13] GRIMMER B., KRIEDER S., RAICU I.: Enabling Dynamic Memory Management Support for MTC on NVIDIA GPUs. EuroSys 2013, 2013. 2

[HRJ*10] HUANG X., RODRIGUES C. I., JONES S., BUCK I., HWU W.-M.: XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology* (Washington, DC, USA, 2010), CIT '10, IEEE Computer Society, pp. 1134–1139. 2, 4, 5

[HRJ*13] HUANG X., RODRIGUES C. I., JONES S., BUCK I., HWU W.-M.: Scalable SIMD-parallel Memory Allocation for Many-core Machines. *The Journal of Supercomputing 64*, 3 (June 2013), 1008–1020. 2, 3

[Kno65] KNOWLTON K. C.: A Fast Storage Allocator. *Communications of the ACM 8*, 10 (Oct. 1965), 623–624. 2, 4

[LC12] LIU R., CHEN H.: SSMalloc: A Low-latency, Locality-conscious Memory Allocator with Stable Performance Scalability. In *Proceedings of the Asia-Pacific Workshop on Systems* (New York, NY, USA, 2012), APSYS '12, ACM, pp. 15:1–15:6. 1

[NBGS08] NICKOLLS J., BUCK I., GARLAND M., SKADRON K.: Scalable Parallel Programming with CUDA. *Queue 6*, 2 (Mar. 2008), 40–53. 1, 2

[NVI10] NVIDIA CORPORATION: *NVIDIA CUDA C Programming Guide 3.2*, November 2010. 2

[NVI12] NVIDIA CORPORATION: *NVIDIA CUDA C Programming Guide 4.2*, April 2012. 4, 6

[SHGV14] SPLIET R., HOWES L., GASTER B. R., VARBANESCU A. L.: KMA: A Dynamic Memory Manager for OpenCL. In *Proceedings of the 7th Workshop on General Purpose Processing Using GPUs* (New York, NY, USA, 2014), GPGPU-7, ACM, pp. 9–18. 2

[SKK*14] STEINBERGER M., KENZEL M., KAINZ B., WONKA P., SCHMALSTIEG D.: On-the-fly Generation and Rendering of Infinite Cities on the GPU. *Computer Graphics Forum 33*, 2 (2014), 105–114. 2

[SKKS12] STEINBERGER M., KENZEL M., KAINZ B., SCHMALSTIEG D.: ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU. In *Innovative Parallel Computing (InPar), 2012* (May 2012), pp. 1–10. 1, 2, 4, 5, 6

[SKL11] SEO S., KIM J., LEE J.: SFMalloc: A Lock-Free and Mostly Synchronization-Free Dynamic Memory Allocator for Manycores. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 2011), PACT '11, IEEE Computer Society, pp. 253–263. 1

[TPO10] TZENG S., PATNEY A., OWENS J. D.: Task Management for Irregular-parallel Workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2010), HPG '10, Eurographics Association, pp. 29–37. 2

[WWWG13] WIDMER S., WODNIOK D., WEBER N., GOESELE M.: Fast Dynamic Memory Allocator for Massively Parallel Architectures. In *Proceedings of the 6th Workshop on General Purpose Processing Using GPUs* (New York, NY, USA, 2013), GPGPU-6, ACM, pp. 120–126. 2, 3, 5, 6