Register Efficient Dynamic 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
xvinkl@fi.muni.cz, havran@fel.cvut.cz

Abstract
We compare five 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 past and we add one with a real workload, where dynamic memory allocation is used in building the k-d 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.

Keywords: dynamic memory allocation, many-core architecture, GPU, CUDA

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 CPUs to be used on GPUs as well. One such technique is the ability to dynamically allocate memory directly from the kernel running on the GPU. This 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, GPU dynamic memory allocation has only recently drawn some interest. The design of a GPU dynamic memory allocator (further referred to as an 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 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 current solutions. In particular, we target allocators that behave well in real workloads, in our case the construction of a k-d tree data structure. In such scenarios, factors like 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 and Section 3 discusses the design goals an allocator targeting many-core hardware should adhere to. The existing allocators included in our comparison are described in more detail in Section 4. In Section 5 we propose our new allocator for GPUs. The evaluation tests used for the comparison are defined in Section 6. Section 7 gives the measurements of the allocators' performance in the evaluation tests. The fragmentation of the proposed allocator is summarized in Section 8. Finally, Section 9 summarizes our findings and presents possibilities for future research.

2. Related Work

CPU memory allocators. The design of allocators for single-core and multi-core systems is a well-researched area. A common technique used in these allocators is to maintain one private heap per thread such as in Berger et al. [BMBW00]. Such techniques do not scale well to many-core architectures with thousands of threads running concurrently, as was demonstrated by Steinberger et al. [SKKS12]. For the GPU hardware used in this paper (GTX Titan Black) there are up to 90 warps (SIMD units) running concurrently and up to 960 warps resident on the multiprocessors at the same time. Using a private heap for each of the warps would consume too much memory or make the per warp heaps extremely small.

Another problem with porting multi-core memory allocators to many-core processors is the use of locks. Even the high performance allocators such as TCMalloc [GM] use locking algorithms when accessing the global heap, e.g. when serving a large allocation. Using locks is infeasible for the high number of threads running concurrently on many-core processors as we demonstrate in this paper.

The lock-free memory allocators [SKL11, LC12] also try to achieve low latency by caching free blocks at multiple levels. This is again less beneficial for GPUs that feature much smaller sizes of L1 and L2 caches, and may even hurt performance because of the high latency of subsequent memory accesses.

GPU memory allocators. 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 toolkit [NBGS08] 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 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), 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 the `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 the GPU memory in CUDA was further researched by Steinberger et al. [SKKS12], who call their allocator `ScatterAlloc`. They ported some of the current allocators designed for the CPU to the GPU and showed that these are not efficient in the context of massively parallel processors. Based on their findings, they set the design goals for a GPU allocator. They compare `ScatterAlloc` to `CudaMalloc` and `XMalloc` showing that hashing significantly reduces the allocation times and thus `ScatterAlloc` outperforms the previous approaches.

Another CUDA allocator, `FDGMalloc` was proposed by Widmer et al. [WWWG13]. They use a two-level allocation scheme as in the paper by Huang et al. [HRJ*13] and target many subsequent allocations. The authors claim that their allocator is faster than `ScatterAlloc` by a factor of 10 to 1000. 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`.

The `Halloc` allocator of Adinetz and Pleiter [AP14] uses a clever hashing function in the spirit of `ScatterAlloc` to significantly reduce the cost of small memory allocations. For a large number of allocating threads, their allocator is supposed to be up to 1000$\times$ faster than `ScatterAlloc`. Memory allocations larger than 3 kilobytes are delegated to `CudaMalloc` similarly to `FDGMalloc`.

Recently, a memory allocator has also been developed for OpenCL [SGS10] 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. Design Goals

A dynamic memory allocator targeting many-core GPU architectures has to adhere to several design goals to be generally applicable and to achieve high performance. We follow the design goals summarized for the `ScatterAlloc` in the paper of Steinberger et al. [SKKS12]. Their summary builds on three categories: **correctness**, **speed**, and **memory overhead** with several design goals falling under these categories. We believe this list is not complete and add new design goals to the list.

Register usage. Apart from the internal and external memory fragmentation and overhead of the allocator itself, on GPUs the number of registers used by the allocator is crucial as well. The registers used up by the allocator may increase the register usage of the entire kernel and thus influence the occupancy of the kernel or the efficiency of the compiled code. Thus, we target solutions that consume as few registers as possible while keeping high performance.

Variability of the allocation requests. For many applications it is sufficient to allocate memory chunks of a single or similar size. This is also usually the case for generic tests for the allocators. Some applications, however, exhibit highly varying sizes of allocation requests. One of such applications is the construction of the *k*-d tree data structure we use in this paper as a real workload. The allocator should be able to serve even such allocation requests efficiently to be generally useful.

Some of the design goals presented in the work of Steinberger et al. [SKKS12] are also not as important on the current generation of hardware as they were on the previous generations.

Scalability. Steinberger et al. try to avoid atomic access to a single location by multiple threads, because it results in linear performance decrease. While this is true in theory, the problem is much less profound in practice. With the upgrade of atomic operations in the Kepler architecture, atomic operations are handled very efficiently. Moreover, because the threads are usually doing computations other than just allocating memory, all threads are seldomly accessing a single location at the same time. This can be observed on the performance of the `AtomicMalloc` allocator that demonstrates worst case atomic operations scalability in both the generic tests and real workloads. Despite this, it is usually one of the fastest allocators even on a GPU with a very high number of concurrently running threads.

False sharing. Accessing data in the same cache line by several processors, is also not much of an issue on current GPUs because the L1 caches are usually incoherent. Thus, no sharing of cache lines occurs and these need not be exchanged between the processors.

4. Existing Allocators

We use five 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 the corresponding papers, these five allocators should include the fastest allocators for GPUs. In this section, we describe in more detail how these allocators operate and highlight their strengths and weaknesses.

4.1. AtomicMalloc

The simplest allocator can be implemented by using a single atomic instruction (see Algorithm 1). This allocator has been hinted at in several publications (e.g. Tzeng et al. [TPO10]), but to the best of our knowledge it has never been formally described or 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.

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.

Algorithm 1: `AtomicMalloc` allocation.

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

4.2. CudaMalloc

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

4.3. 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 of 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, which 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 pages already locked and restarts the search in another location. To lower the probability of restarting, only a single thread may try to allocate 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.

4.4. 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 an 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.

4.5. Halloc

The last allocator that we use is the Halloc allocator, recently proposed by Adinetz and Pleiter [AP14]. This allocator splits the assigned size of the dynamic memory pool into two parts. One part is served by CudaMalloc and handles allocations larger than 3 kilobytes. The other part is preallocated in the GPU memory in the form of slabs. These slabs of fixed size handle small memory allocations. The slabs are assigned during runtime to a particular size of allocation requests. When the slab is freed it is allowed to be reinitialized for another size of requests. The information about the usage of each slab is stored in a bitmap and is updated using atomic operations. Hashing is utilized in finding a free block of memory in the bitmap to reduce the conflicts of atomic operations.

5. 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, FDGMalloc and Halloc are biased towards small and repetitive allocations. In this section, we propose allocators that alleviate some of these limitations.

5.1. 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).

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;
```

If there is enough contiguous memory, 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*.

The wrap may result in 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 only for brief periods, this allocator can provide the performance of AtomicMalloc with significantly reduced 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.

5.2. CircularMalloc (CMalloc)

Our second proposed allocator organizes the memory pool as a singly linked list. We 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 want to design an allocator that is also able to 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 chunk would require serialization of the threads locking the chunk's flag as in AWMalloc but with a much

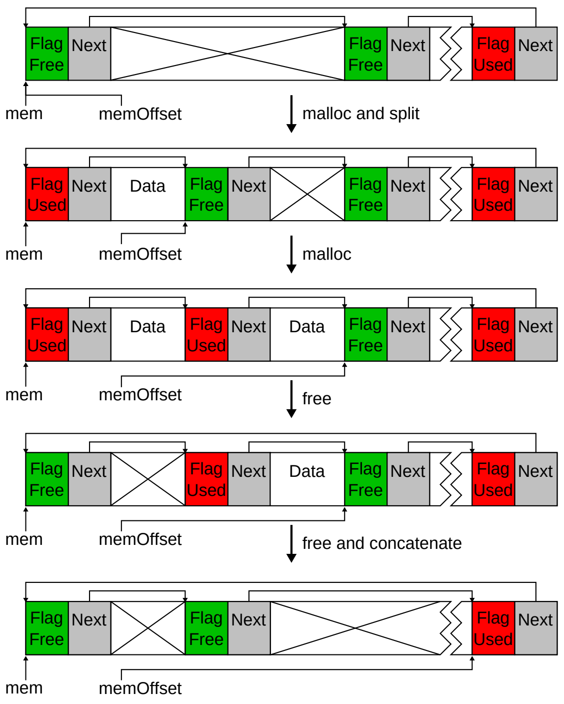


Figure 1: `CMalloc` allocation and deallocation. The header for each allocation consists of the flag and the next pointer. The memory pool is pre-split into arbitrarily sized chunks and the end of the memory pool is marked with a header that stores a used flag and a pointer to the beginning of the memory 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.

slower time for a single allocation. An example of allocation and deallocation using `CMalloc` is shown in Figure 1.

We propose pre-splitting 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):

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

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. In other words, for every $k = 2^x$ there are k consecutive chunks of size R/k. The memory not used by the heap forms the last chunk. This pre-splitting may be viewed as the partitioning of memory after a single allocation request of the smallest chunk size in the buddy memory allocator [Kno65]. In our tests, such division in most cases provided the best performance while having high flexibility in the size of the allocation requests that can be served by the allocator.

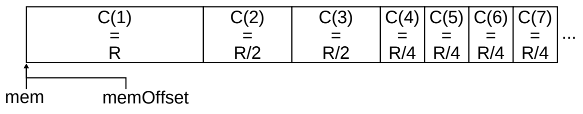


Figure 2: Memory pool pre-split into a binary heap.

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 the 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 the 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. Once the newly created chunk header has been 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 an 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 possibly be 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. The concatenation of chunks cannot cause the *offset* variable in `mallocCMalloc` to point to a header that was already overwritten by something else. This is because the allocating thread cannot be stalled for long enough to let this overwrite happen. After the free operation, the *memOffset* is moved past the concatenated chunk and the following allocations will thus take place there as well. It would take a significant number of global memory accesses to move the allocation offset to the beginning of the concatenated chunk.

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.

Algorithm 3: `CMalloc` allocation and deallocation.

```
mallocCMalloc(size) begin
    offset ← memOffset;
    size ← size + HEADER_SIZE;
    while (true) do
        lock ← atomicCAS(mem[offset], Free, Used);
        next ← mem[offset + NEXT_OFS];
        csize ← next − offset;        // Chunk size
        if (lock = Free) then
            if (size ≤ csize) then
                break;
            mem[offset] ← Free;
        offset ← next;
    newNext ← next;
    if (size + HEADER_SIZE ≤ csize and
        size·MAX_FRAG ≤ csize) then
        newNext ← offset + size;
        mem[newNext] ← Free;
        mem[newNext + NEXT_OFS] ← next;
        __threadfence();
        mem[offset + NEXT_OFS] ← newNext;
    memOffset ← newNext;
    return mem + offset + HEADER_SIZE;

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

Although list-based allocators have previously been tested [HRJ*10, SKKS12], our approach uses 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 advances in graphics hardware, mainly the addition of hardware cache hierarchies and improved atomic instructions.

5.3. Circular Fused Malloc (CFMalloc)

Given that the flag for each chunk holds just the free/used information, it can be contained in a single bit of data. If fewer than 2^{31} words need to be allocated, 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. Such fusion supports up to 8 GB of allocable space with 4-byte words, which is usually sufficient even on current high-end GPUs (12 GB of RAM) because not all of the device memory needs to be dynamically allocable.

5.4. 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 reduces the number of conflicts in 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 the 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.

6. Evaluation Tests

To test the allocators, we implemented three generic evaluation tests introduced in previous papers on dynamic memory allocation on GPUs. In addition to these, we 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 load 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 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 (k-d tree [Hav00,WH06,ZHWG08]) on the GPU. In this data structure, geometric primitives straddling the splitting plane may be duplicated in both the left and right children of the split node. To solve this duplication of geometric primitives, two new chunks of memory have to be allocated to hold the geometric primitives in the left and right child nodes. This scenario is highly challenging since a large number of geometric primitive arrays of varying size needs to be allocated and deallocated during the build and the order of the allocations is unknown. In our k-d tree builder a warp is the unit of computation and thus only a single thread of a warp dynamically allocates memory.

Using dynamic memory allocation directly on the GPU slightly increases the footprint of the memory pool compared to static allocation of GPU memory independently for each level, as done by, e.g., Wu et al. [WZL11]. The memory pool for dynamic allocation has to be large enough to accommodate the allocation requests of all currently processed nodes, not just the ones of the nodes being split on the current level. On the other hand, building the entire k-d tree in a single kernel significantly decreases the kernel launch overhead and thus achieves lower build times than even hybrid CPU-GPU builders [RPC12]. The single kernel k-d tree builder is 25 – 45% faster than the reference method [VHBS14].

Test considerations. We decided to let a single thread of each warp allocate memory in all of these evaluation tests. The techniques for coalescing memory allocations inside a SIMD unit are well researched and used in all of the previously published methods for dynamic memory allocations on the GPU. The same or similar code may thus be used for the other allocators with the same benefit of decreasing the number of allocation requests. For each of the compared allocators we have implemented one variant with and without coalescing. In our tests where just one thread is allocating memory, these two variants do not differ in their performance because the coalescing does not incur slow global memory accesses. However, they can differ in the number of used registers (see Table 1).

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`, `FDGMalloc` and `Halloc` are doing so internally [SKKS12, WWWG13, AP14].

7. Results

We evaluated the allocators on a PC with Intel Core i7-2600, 8 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 measured 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.

Allocator	Register usage	
	Uncoalesced	Coalesced
`AtomicMalloc*`	4	9
`CudaMalloc`	6	17
`ScatterAlloc`	38	45
`FDGMalloc`	23	26
`Halloc`	48	56
`AWMalloc*`	6	9
`CMalloc`	18	18
`CFMalloc`	14	16
`CMMalloc`	18	22
`CFMMalloc`	16	21

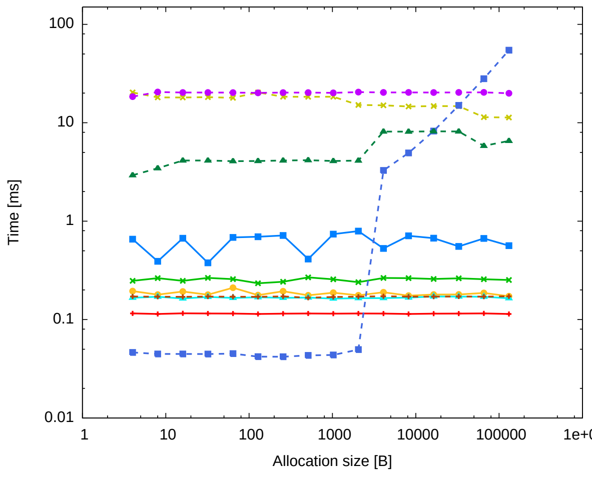
Table 1: Register usage of the allocators with and without support for coalescing allocation requests inside a warp. For each allocator the number of registers used in a kernel consisting of a single allocation and deallocation is reported. The allocators marked with an asterisk contain no deallocation code.

The setting for all generic tests is given in Table 2. The number of launched threads is $\#Blocks \times 256$, where $\#Blocks$ usually equals 120 to fully saturate the GPU. Only the first thread of each warp however allocates memory, thus the number of allocations is 32 times smaller. This reduction of allocation requests is possible even for real applications through coalescing of the allocation requests.

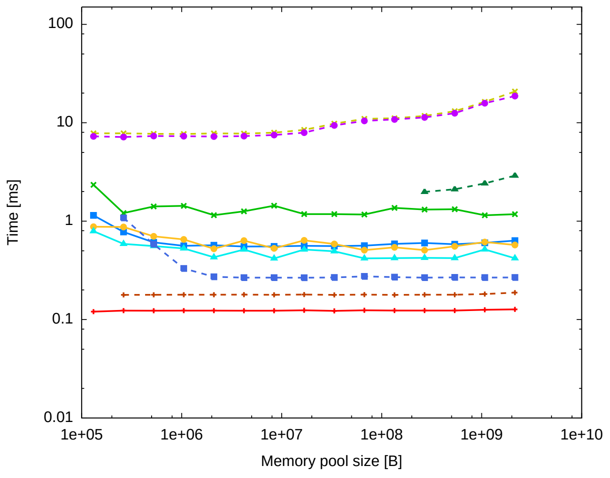
[AD] Alloc Dealloc. First, we tested the influence of increasing the size of memory requested by the threads (see

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)

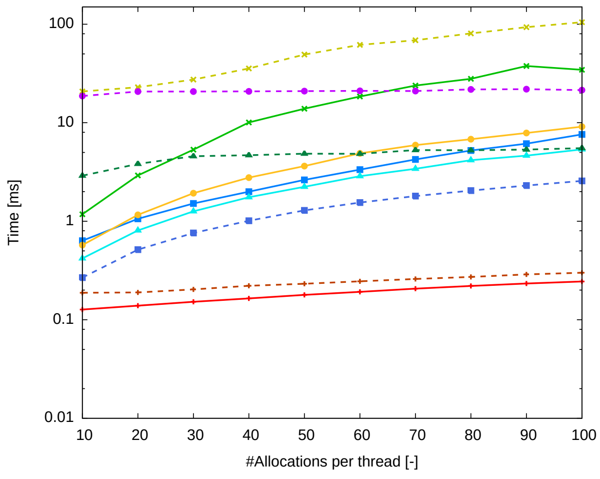
Table 2: The setting of our tests. The number of allocations (and deallocations) performed by all threads is $\#Blocks \times 8$ (warps per block) $\times \#Iters$ except for the Probability test where the number of allocations is 1450 and the number of deallocations is 986 when launching 120 blocks.



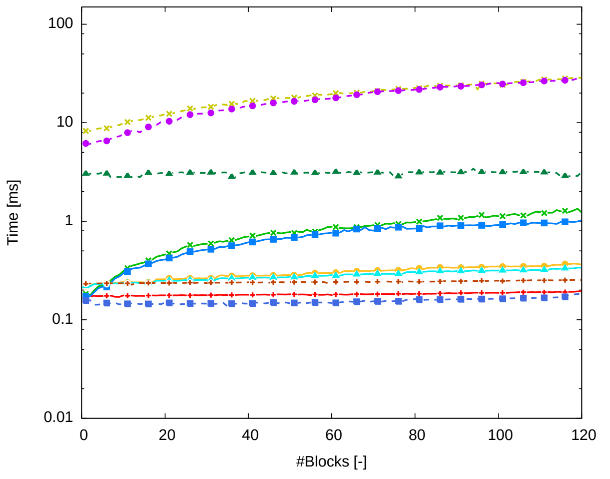
(a) Alloc Dealloc test.



(b) Alloc Cycle Dealloc test.



(c) Alloc Cycle Dealloc test.



(d) Probability test

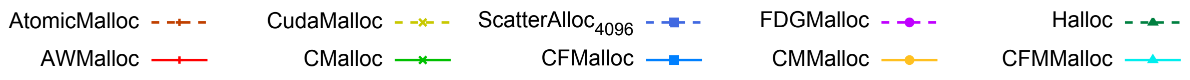


Figure 3: Graphs showing the properties of the allocators in various benchmarks. (a) Alloc Dealloc test — increasing the size of memory requested in each allocation, (b) Alloc Cycle Dealloc test — increasing the size of the memory pool, (c) Alloc Cycle Dealloc test — increasing the number of allocations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

Figure 3(a)). CudaMalloc and FDGMalloc are by far the slowest allocators in this test, with FDGMalloc being slower than CudaMalloc. This is because 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. Similar behavior can be observed for the Halloc since for larger allocations the CudaMalloc is used instead of hashing.

[ACD] Alloc Cycle Dealloc. Since the performance of ScatterAlloc depends on the size of its pages, we 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 (that inludes a lock), the length 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 greater than 64 MB the test fails completely.

Figure 3(b) shows the dependence of the length 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 memory pool than for larger ones as conflicts of atomic instructions are more likely to occur due to imperfect hashing. For `Halloc`, with the default size of the slabs the memory pool has to be larger than 256 MB, otherwise the memory pool fails to initialize. For smaller slab sizes, smaller memory pools are possible, but at the cost of decreased performance. 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 allocators other than `CudaMalloc`. However, `FDGMalloc` may use any of the faster allocators for the allocation of superblocks, while maintaining constant performance in iterative allocations. `Halloc` also deals with subsequent allocations well, eventually surpassing the `CMalloc` and its variants.

[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`, `ScatterAlloc` and `Halloc` having an almost flat slope.

The performance of `ScatterAlloc` can approach that 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 by 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 1.57× to 2.09× slower as shown in Figures 3(c) and 3(d). This is likely due to the conflicts of atomic operations.

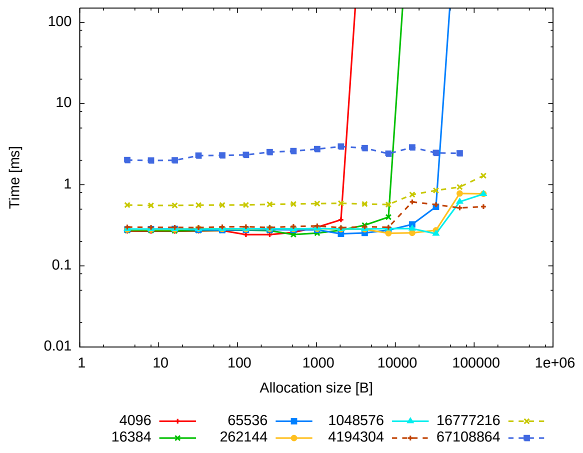


Figure 4: Dependence of `ScatterAlloc` on the page size (the individual curves) in the Alloc Cycle Dealloc test. The exact setting of the test is shown in Table 2.

[DS] Data structure build. Generic tests where only allocation and deallocation operations are performed and all threads allocate memory at the same time may not represent real workloads well. For this reason, we also compare the *k*-d tree build times when using different allocators (see Table 3). The size of the memory pool is set to 200 times the size of the root node, which is sufficient to run the build even with `AtomicMalloc`. `FDGMalloc` is excluded from this test because it can only deallocate all of the memory at the end of the computation. Moreover, the memory 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 *k*-d tree build time.

From Table 3 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. However, this does not seem to be the main bottleneck in the *k*-d tree build, as demonstrated by `Halloc` which consumes even more registers. The influence of regis-

	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	4.7	5.42	6.3	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	2,982K
$\bar{X}_{\|alloc\|}$ $[B]$	45.8	81.8	100.8	101.8	79.8	167.8	51.9	44.5	40.2	45.7	113.2
$\sigma_{\|alloc\|}$ $[B]$	335.6	594.4	734.7	791.1	638.1	989.5	458.1	388.3	362.1	394.1	721.1
Allocator						T_{build} $[ms]$					
`AtomicMalloc*`	8.55	18.40	16.0	27.3	46.1	31.2	70.1	384.7	944.7	2243.2	305.2
						Slowdown $[-]$					
`CudaMalloc`	5.45	2.95	2.50	3.19	4.11	2.59	4.91	20.62	18.72	11.17	3.80
`ScatterAlloc`$_{4096}$	1.66	1.73	1.50	1.80	2.16	2.47	2.24	1.75	2.05	0.94	4.92
`ScatterAlloc`$_{8192}$	1.73	1.72	1.32	1.46	2.04	1.86	2.73	8.72	1.76	1.03	2.91
`ScatterAlloc`$_{16384}$	1.83	2.02	1.64	2.09	2.71	1.83	3.58	6.75	6.57	4.75	1.97
`Halloc`	**1.04**	**1.10**	**1.02**	**1.11**	**1.22**	**1.13**	1.39	**1.15**	**1.06**	1.12	1.41
`FDGMalloc*`	–	–	–	–	–	–	–	–	–	–	–
`AWMalloc*`	1.00	0.88	0.94	0.87	0.96	0.83	1.00	0.90	1.07	0.96	0.95
`CMalloc`	1.48	1.24	1.25	1.28	1.32	1.25	1.25	1.39	1.21	0.99	**1.19**
`CFMalloc`	1.14	1.20	1.16	1.23	1.28	1.22	**1.23**	1.36	1.16	**0.92**	1.22
`CMMalloc`	1.30	1.23	1.26	1.24	1.33	1.30	1.36	1.42	1.49	0.97	1.21
`CFMMalloc`	1.27	1.25	1.24	1.25	1.32	1.28	1.34	1.25	1.24	0.97	1.24

Table 3: The allocation properties and build times of k-d 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 `AtomicMalloc` the build times of k-d 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 typeset in boldface. The allocators marked with an asterisk are not full allocators and are not considered as being the fastest.

ter utilization on the allocators' performance is hard to evaluate precisely since artificially lowering register utilization influences both the allocator and the k-d tree build algorithm.

The `CMalloc` and `Halloc` which were not among the fastest allocators in the generic tests, performed very well for the k-d tree build, with `Halloc` usually being faster for the scenes in Table 3. This is caused by the allocation properties of the build, with most of the allocation requests being issued for nodes near the leaves. These requests are usually small, playing on the strengths of `Halloc`. This is supported by the fact that the minimum size of the memory pool for `Halloc` is 256 MB, which is more than for the other allocators on most of the scenes (up to 336K triangles). A larger memory pool makes it easier for `Halloc` to find a free memory chunk using hashing. For larger scenes, the same size of memory pool is used for all allocators. In our measurements `Halloc` also had larger variance of build times, being sometimes faster and sometimes slower than `CMalloc` for the same scene. We think this may be caused by the hashing used in `Halloc`. The reported times show its faster performance.

For the scenes in Table 4, 20 times the size of the root node is used as the memory pool size, requiring the freed memory to be reused. Moreover, there is a higher number of large allocation requests because the scenes are larger. In such conditions, `CMalloc` becomes the fastest allocator. Notice that `AtomicMalloc` and `AWMalloc` cannot be used to build the k-d trees for these scenes since the size of the memory pool is insufficient for their operation, and a larger memory pool cannot be used due to the size of the GPU DRAM.

Contrary to the generic tests, `CMMalloc` and `CFMMalloc` allocators are often slower than `CMalloc` and `CFMalloc` when used in the k-d tree build. In this test, the warps are not requesting memory allocations at the same time during the

	Scene				
	House 3×3	Asian Dragon	San Miguel	MPII subset	Power-plant
N_{tris} $[-]$	3,275K	7,219K	7,881K	10,762K	12,749K
N_{ref} $[-]$	9.37	7.16	7.03	6.67	5.99
N_{alloc} $[-]$	13M	25M	17M	18M	15M
$\bar{X}_{\lvert alloc \rvert}$ $[B]$	55.6	46.0	78.6	101.7	144.3
$\sigma_{\lvert alloc \rvert}$ $[B]$	452.1	442.5	581.9	723.5	799.8
Allocator	T_{build} $[s]$				
`Halloc`	4.94	20.83	7.94	9.04	8.45
	Slowdown $[-]$				
`CudaMalloc`	2.38	2.33	2.28	2.02	1.53
`ScatterAlloc`$_{4096}$	3.06	1.04	2.66	8.58	5.81
`ScatterAlloc`$_{8192}$	2.52	0.61	0.92	2.97	2.14
`ScatterAlloc`$_{16384}$	3.26	1.65	4.30	1.10	2.53
`CMalloc`	**0.64**	**0.43**	**0.63**	**0.64**	0.63
`CFMalloc`	0.65	0.47	0.69	0.67	0.63
`CMMalloc`	0.65	0.48	0.67	0.65	0.66
`CFMMalloc`	0.67	0.53	0.64	0.67	**0.61**

Table 4: The allocation properties and build times of k-d trees when using the allocators on large scenes. The legend is the same as for Table 3. For `Halloc` the build times of k-d trees are given in seconds 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.

build, canceling the usefulness of multiple offsets into the memory pool.

We 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 in fewer conflicts of atomic operations on a shared variable. In the generic tests the GTX 680 was almost twice as fast for all of the allocators except `ScatterAlloc` which is less prone to these conflicts.

8. Fragmentation

For our newly proposed allocators we also analyze their memory properties, in particular, their memory fragmentation. We do not analyze memory fragmentation for the other allocators because this would require changes in their source codes. In particular, this is impossible for `CudaMalloc` for which the source code is not available. We follow the definitions provided in the paper by Steinberger et al. [SKKS12].

We suppose the memory is split into a set of regions R and define two functions, *alloc* and *size*. The function $alloc : R \to \mathbb{N}$ maps a region r to the size of the memory request that caused its allocation, or 0 for a free region. This function divides the set R into two disjoint subsets: $A = \{r \in R \mid alloc(r) \neq 0\}$ is the set of all allocated regions and $F = R \setminus A$ is the set of all free regions. The other function, $size : R \to \mathbb{N}$, returns the size of a region r.

Internal fragmentation and external fragmentation is defined as:

$$F_{internal} = \frac{1}{|A|} \cdot \sum_{r \in A} \frac{size(r) - alloc(r)}{size(r)}, \qquad (2)$$

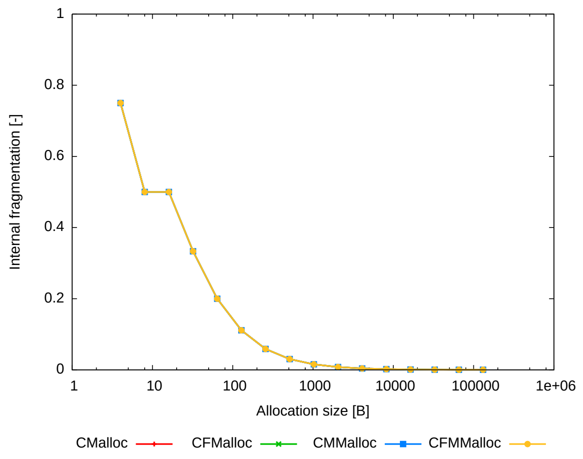
$$F_{external} = 1 - \frac{\max_{f \in F} size(f)}{\sum_{r \in F} size(r)}. \qquad (3)$$

The internal fragmentation is the measure of the unusable memory caused by the memory alignment requirements (16 bytes in our measurements) or the allocator design. The $F_{internal}$ may thus vary with the allocation pattern of the application. The external fragmentation is the measure of the ability of the allocator to serve large allocation requests. If the memory is too fragmented due to the allocator's design or because of the strategy used to find the free memory region, the allocator may be unable to serve such requests. The value of $F_{external} = 0$ means that all of the free memory can be allocated in a continuous block of memory while a value close to 1 means that the memory is heavily fragmented.

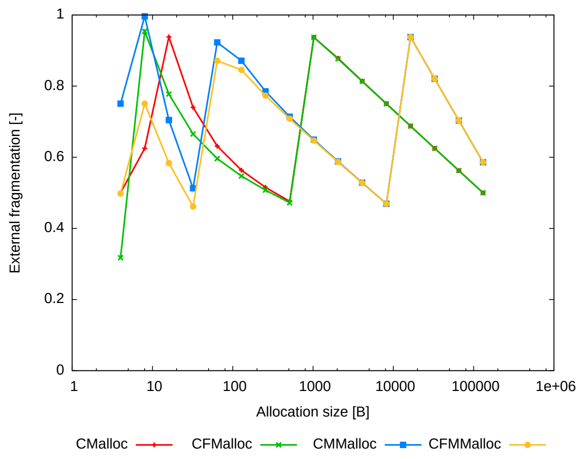
`AWMalloc` has by definition a very low internal fragmentation caused only by the memory alignment, and the sole overhead is the 4 byte atomic counter *memOffset*. The external fragmentation during the launch of the kernel using this allocator depends on the order of the allocation and deallocation requests. At the end of the kernel launch it is 0, considering all allocated memory has been freed.

The memory properties for `CMalloc` and its variants are more interesting. The memory overhead is the 4-byte *memOffset* and N headers, which divide the memory pool. The size of the header is 8 bytes for `CMalloc` and `CMMalloc` and 4 bytes for `CFMalloc` and `CFMMalloc`. To measure the external fragmentation, we scan the memory pool when the test is finished. While all of the allocations are freed at this point in our tests, the memory pool is still fragmented into smaller regions by means of the headers.

The internal fragmentation for `CMalloc` and its variants in the Alloc Dealloc test when changing the size of the allocation requests is given in Figure 5(a). It is the same for all variants of `CMalloc` since they follow the same design regarding memory alignment and the maximum internal fragmentation allowed. The internal fragmentation for the first data point is very high (0.75) because only 4 bytes are used of the minimum allocable chunk of 16 bytes. With increasing size of the allocation requests, the header size and the memory alignment become insignificant and the internal fragmentation quickly drops towards zero. This is consistent with the



(a) Internal fragmentation.



(b) External fragmentation.

Figure 5: Dependence of (a) internal and (b) external fragmentation of `CMalloc` and its variants on the allocation size in the Alloc Dealloc test. The internal fragmentation is the same for all variants of `CMalloc` since they follow the same design regarding memory alignment and the maximum internal fragmentation allowed.

design targeting large allocations. There is no difference between the fused and unfused variants because it is usually masked by the memory alignment. The external fragmentation in Figure 5(b) is much more varied, ranging from 0.3 up to almost 1. Still, it is usually much lower than for the methods that uniformly pre-split the memory pool such as `ScatterAlloc` or `Halloc`. We have verified this by using a pre-splitting strategy with uniform sizes of the chunks for `CMalloc` as well. Using this pre-splitting the external fragmentation was consistently very close to 1.

For the *k*-d tree build test the internal fragmentation is always lower than 0.01, confirming the allocator's benefit for large allocations. The external fragmentation is usually between 0.94 and 0.995, because the build finishes with a large number of small memory allocations that are not recursively concatenated.

For both generic and real workloads, the external fragmentation may be improved by a different pre-splitting strategy, e.g. by leaving a large free chunk of memory at the end of the memory pool, or by recursive concatenation of free chunks as in the buddy memory allocator [Kno65].

9. Conclusions and Future Work

In this paper we have proposed a new dynamic memory allocator designed for GPUs and its variants, and compared them to five existing dynamic memory 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. `Halloc` should be used when there is a very large number of allocating threads or successive allocation requests and `CMalloc` or its variants should be used if the allocation properties are unknown.

We showed that the limitations of existing dynamic memory allocators, which may not manifest in generic tests, cause significant slowdown in real workloads. For applications with large variability in allocation sizes or high regis-

Allocator	Register usage	Variable requests	Succ. re-quests	Scaling	Overall Speed
`CudaMalloc`	+	+	0	0	-
`AtomicMalloc*`	+	+	+	+	+
`CMalloc`	+†	+	0	0‡	+
`ScatterAlloc`	-	-	0	+	0
`FDGMalloc*`	0	+	+	0	-
`Halloc`	-	0	+	+	0

Table 5: Property summary for the dynamic memory allocators. The '+' symbol represents that the allocator performs well in the property, '0' is for average performance and '-' for poor performance. The dynamic memory allocators marked with an asterisk are not full memory allocators since they cannot deallocate individual chunks of memory. † '+' or '0' for different variants. ‡ '+' when not all allocations take place at the same time or multiple offsets to the pool are used for different multi-processors.

ter utilization, our proposed simple dynamic memory allocator `CMalloc` and its variants are capable of outperforming state-of-the-art dynamic memory allocators. For the memory allocation/deallocation pattern of the parallel *k*-d tree building algorithm, the speedup computed from the whole running time when using `CMalloc` compared to `ScatterAlloc` with tuned page size is between 1.4× and 4.6× on large scenes. In addition, our `CMalloc` does not need any parameters to be set. Compared to `Halloc` our allocator is between 7.7× and 49.0× faster on large scenes, but up to 13% slower on small scenes.

The characteristics of the dynamic memory allocators compared in this paper are summarized in Table 5 for both generic and real workloads. `CMalloc` (the method proposed in this paper) is a good all-round allocator.

To ease the use of the new dynamic memory allocators, we provide their source codes in the form of a library that can be accessed at `http://dcgi.fel.cvut.cz/projects/CMalloc`. In future work we would like to investigate the efficiency of recursive concatenation of free chunks in `CMalloc` and hybrid allocation strategies. The techniques typically employed on CPUs may behave quite differently under the memory hierarchy of GPUs.

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, Guillermo M. Leal Llaguno for the San Miguel model, Vlastimil Havran et al. for the MPII model [HZDS09], the University of North Caroline at Chapel Hill (UNC) for the Powerplant model, and the 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

[AP14] ADINETZ A. V., PLEITER D.: Halloc: a High-Throughput Dynamic Memory Allocator for GPGPU Architectures, March 2014. `https://github.com/canonizer/halloc`. 2, 4, 7

[BMBW00] BERGER E. D., MCKINLEY K. S., BLUMOFE R. D., WILSON P. R.: Hoard: A Scalable Memory Allocator for Multithreaded Applications. *SIGPLAN Not. 35*, 11 (Nov. 2000), 117–128. 2

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

[GM] GHEMAWAT S., MENAGE P.: Tcmalloc: Thread-caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`. 2

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000. 7

[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, 6

[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

[HZDS09] HAVRAN V., ZAJAC J., DRAHOKOUPIL J., SEIDEL H.-P.: *MPI Informatics Building Model as Data for Your Research*. Research Report MPI-I-2009-4-004, MPI Informatik, Saarbruecken, Germany, Dec 2009. 13

[Kno65] KNOWLTON K. C.: A Fast Storage Allocator. *Communications of the ACM 8*, 10 (Oct. 1965), 623–624. 2, 5, 6, 12

[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. 2

[NBGS08] NICKOLLS J., BUCK I., GARLAND M., SKADRON K.: Scalable Parallel Programming with CUDA. *Queue 6*, 2 (Mar. 2008), 40–53. 2, 3

[NVI10] NVIDIA CORPORATION: *NVIDIA CUDA C Programming Guide 3.2*, November 2010. `https://developer.nvidia.com/cuda-toolkit-32-downloads`. 2

[NVI12] NVIDIA CORPORATION: *NVIDIA CUDA C Programming Guide 4.2*, April 2012. `https://developer.nvidia.com/cuda-toolkit-42-archive`. 5, 7

[RPC12] ROCCIA J.-P., PAULIN M., COUSTET C.: Hybrid CPU/GPU KD-Tree Construction for Versatile Ray Tracing. In *Eurographics (Short Papers)* (2012), Eurographics Association, pp. 13–16. 7

[SGS10] STONE J. E., GOHARA D., SHI G.: OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering 12*, 3 (May 2010), 66–73. 2

[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* (Piscataway, NJ, USA, May 2012), IEEE, pp. 1–10. 2, 3, 6, 7, 11

[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. 2

[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. 3

[VHBS14] VINKLER M., HAVRAN V., BITTNER J., SOCHOR J.: Parallel On-Demand Hierarchy Construction on Contemporary GPUs. Transactions on Visualization and Computer Graphics (submitted minor revision), 2014. 7

[WH06] WALD I., HAVRAN V.: On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2006* (Washington, DC, USA, sep. 2006), IEEE Computer Society, pp. 61–69. 7

[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, 6, 7

[WZL11] WU Z., ZHAO F., LIU X.: SAH KD-tree Construction on GPU. In *Proceedings of HPG 2011* (New York, NY, USA, 2011), ACM SIGGRAPH/Eurographics, pp. 71–78. 7

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time KD-tree Construction on Graphics Hardware. In *SIGGRAPH Asia 2008* (New York, NY, USA, 2008), ACM, pp. 126:1–126:11. 7