ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU

Markus Steinberger, Michael Kenzel, Bernhard Kainz, Dieter Schmalstieg
Institute for Computer Graphics and Vision
Graz University of Technology
{steinberger|kenzel|kainz|schmalstieg}@icg.tugraz.at

ABSTRACT

In this paper, we analyze the special requirements of a dynamic memory allocator that is designed for massively parallel architectures such as Graphics Processing Units (GPUs). We show that traditional strategies, which work well on CPUs, are not well suited for the use on GPUs and present the thorough design of *ScatterAlloc*, which can efficiently deal with hundreds of requests in parallel. Our allocator greatly reduces collisions and congestion by scattering memory requests based on hashing. We analyze *ScatterAlloc* in terms of allocation speed, data access time and fragmentation, and compare it to current state-of-the-art allocators, including the one provided with the NVIDIA CUDA toolkit. Our results show, that *ScatterAlloc* clearly outperforms these other approaches, yielding speed-ups between 10 to 100.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Allocation / deallocation strategies*

Keywords

dynamic memory allocation, GPU, massively parallel, hashing

1. INTRODUCTION

Dynamic memory allocation is an indispensable feature of modern operating systems and virtually every computer program depends on this feature to allow for a dynamic response to varying inputs. Consequently, almost every general purpose programming language provides some mechanism to create new objects at runtime. Efficient dynamic memory allocation became more demanding with the advent of multi-core CPUs. Since system memory is a shared resource, conflicts during memory allocation are an issue. To avoid conflicts, synchronization among multiple cores is required. However, this synchronization requirement defines a serious bottleneck for many applications [17].

Besides the development of multi-core CPUs, the use of massively parallel hardware architectures like *Graphics Processing Units* (GPUs) for general purpose applications became more and more popular. For these kinds of architectures, dynamic memory allocation is an even greater challenge. While current consumer CPUs feature between four to eight cores, current graphics cards are shipped with several hundred cores. In addition to the much larger number of cores competing for memory, the architecture's high latency for accessing global shared memory complicates the development of an efficient dynamic memory allocator.

The most recent NVIDIA GPU architecture, *Fermi*, alleviates the problem of high latency for global shared memory access by introducing a cache [18]. For this architecture, NVIDIA's *Compute Unified Device Architecture (CUDA)* programming model supports dynamic memory allocation in device code [19]. Dynamic memory allocation for the stream programming model opens up completely new possibilities for GPU programming.

Examples for an efficient use of this new feature range from image detectors, which store local descriptors of varying size, over parallel IP-packet stream analysis, which dynamically generates alerts, to weather simulations, for which pressure area descriptors are dynamically created. Previously, organizing vast numbers of differently sized objects at runtime was only possible using parallel reduction summations (*scan*) [7]. The *scan*-method requires multiple kernel launches: One initial kernel launch to analyze the input data and write the number of required bytes for each data element into global memory, followed by the scan kernels. Subsequently, a final kernel launch is required to write the resulting data. Compared to this procedure, a single call of a dynamic memory allocator function is very likely to be more efficient in terms of programming effort and execution time.

Although NVIDIA makes their dynamic memory allocator available for use within the CUDA toolkit, its internals remain undisclosed. Furthermore, our tests have revealed that their current implementation is unreliable under heavy load. Literature on dynamic memory allocation on massively parallel architectures is rare. The only published and especially for the use on GPUs optimized dynamic memory allocator is *XMalloc* [8]. Therefore, we see an urgent need for further research in this area.

With this work, we want to form a solid base for the design of dynamic memory allocation on massively parallel architectures such as GPUs. To achieve this, we designed and implemented *ScatterAlloc* and give detailed information about its inner structure, performance and implementation. Our main contributions are:

- A discussion of necessary design goals which arise from the architecture of current GPUs. We point out special problem cases and state additional challenges for the implementation of *ScatterAlloc* in Section 3.
- A way to adjust currently available *CPU* strategies for the use on GPUs. This serves as a baseline for a direct comparison to *ScatterAlloc* and forms another reliable allocator in Section 4.
- A design of the novel allocator *ScatterAlloc*, targeting massively parallel execution in Section 5.
- A detailed description of the implementation of *ScatterAlloc* with pseudo-code examples in Section 6.
- A set of tests to evaluate the suitability of a dynamic memory allocator for the stream processing model in Section 7.
- A comparison of the presented allocators with the allocator that is provided with the current CUDA toolkit and *XMalloc*. We

show that *ScatterAlloc* is approximately 100 times faster than the CUDA toolkit allocator and up to 10 times faster than *XMalloc* in Section 8.

2. RELATED WORK

Dynamic memory allocation is one of the very basic operations in Computer Science. While a lot of work has been done on the CPU side to gain more performance and to adapt the memory management mechanisms to newly available hardware features, only little research has been done on the GPU side. Therefore, we summarize first the most important memory allocation methods for the traditional CPU areas in Section 2.1 and then focus on the differences between the hardware architectures, which have to be considered when building a dynamic memory management system for the GPU, in Section 2.2.

2.1 Dynamic memory allocators on the CPU

While earlier programming languages, like for example early versions of Fortran, supported only static memory allocation, modern programming languages support multiple allocation schemes ranging from stack allocation to fine-tuned heap allocators provided by the programming environment itself. Besides optimized standard methods, which can be found in modern high-level programming languages, several researchers proposed improvements to these methods. *Doug Lea's Malloc* [13] and a fast multi-threaded version, ptmalloc [5], are well-known examples of such algorithms. They are currently used in the *GNU C++ library*. A good overview and comaprison of such methods is given by Wilson *et al.* [25]. As they discuss, the primary goal of early allocation schemes is to minimize memory overhead.

With the introduction of multi-core CPUs, traditional memory allocation turned out to be a serious bottleneck when multiple cores try to allocate memory in parallel. David Nicol showed the drawbacks of state-of-the-art memory allocation when it comes to highly parallel and frequent dynamic memory allocation during simulations [17]. To overcome these problems, *parallel heap* implementations have been introduced by Häggander and Lundberg [6]. With the *Hoard* system by Berger and colleagues [2] a reliable and fast solution for multi-threaded dynamic memory allocation has been found. These systems basically combine one global heap with per-processor (*i.e.*, per-thread) heaps for minimal memory consumption and low synchronization costs. However, heap manipulation in such systems often requires locks and atomic operations because threads cannot be sure on which processor they are executed. Altering the Hoard approach leads to *'mostly lock-free malloc'* [4], which requires only one heap per CPU. *Multi-Processor Restartable Critical Sections* as introduced with *'mostly lock-free malloc'*, allowed a speed-up by a factor of ten for selected applications. Later, *'scalable lock-free dynamic memory allocation'* [15] allowed the complete removal of locks. This allocator uses only atomic operations for dynamic memory allocation and is therefore "immune to deadlocks regardless of scheduling policies and provides async-signal-safety, tolerance to priority inversion, kill-tolerance, and preemption-tolerance, without requiring any special kernel support or incurring performance over-head" [15]. A recent improvement of lock-free dynamic memory allocation can be found in the *McRT-malloc* approach [9], which is designed for the use with *software transactional memory (STM)*. Compared to the above-mentioned approaches, this allocator also avoids atomic operations in most cases and detects balanced transactional allocations to avoid space blowup. Most recent state-of-the-art memory allocators for multi-core CPUs still use one local heap per thread [22, 24].

All of these methods have been designed for systems with low or medium degrees of parallelism and do not scale well to massively parallel computation as realized on modern GPUs. Many approaches require one heap per processor or thread which can work well for a limited number of cores. However, GPUs provide an order of magnitude more processing units than any modern CPU, therefore one heap per processor would simply introduce too much overhead. Furthermore, fundamental differences between the hardware architectures prevent the direct application of the above-mentioned methods. These differences are briefly outlined in the next section.

2.2 GPU architecture

In contrast to CPUs, GPUs consist of stream processors. Since stream processors heavily rely on parallelism, operations that require serialization can significantly hurt performance. This problem becomes particularly significant if more than one processor tries to write the same memory address. In case of such memory collisions, the extensive use of mutual exclusive locks and atomic operations is required. A good overview over the main features of stream architectures and their memory allocation and control schemes is given by Ahn [1].

The degree of parallelism of modern GPUs is rarely seen, even in other comparable stream-processor based architectures. Because the single instruction, multiple data (SIMD) model of GPUs in fact forces many stream processors to run the same code in parallel, locking, as it is heavily used for CPU based dynamic memory allocation methods, defines a problem. And the overhead of inter-core communication through main memory becomes a severe bottleneck as shown by [26] and [12].

Specialized algorithms use lock-free queues and work stealing [3] to overcome the lack of suitable dynamic memory allocation on the GPU. However, their overhead makes them only applicable for applications with high computational demands and low parallelism, such as the construction of hierarchical data structures [12].

Furthermore, under certain circumstances, locks might not even work correctly on SIMD architectures such as modern GPUs. CUDA, for example, assigns blocks of threads to a multi processor, which organizes their execution in so called *warps*. Diverging threads in a *warp* are only marked inactive while the other branch executes. Thus, threads acquiring a lock might get caught up in the inherent busy–wait spin–lock, which can prevent the one thread holding the lock from proceeding. Such problems are not only to be found in the NVIDIA Tesla architecture [14], but also in Intel's experimental Larrabee processor [23].

To the best of our knowledge, the only for GPUs optimized allocator is *XMalloc* [8]. To support a faster reuse of memory blocks, *XMalloc* stores freed blocks in queues and serves allocation requests from these queues before new memory is used. Furthermore, *XMalloc* introduces an optimization for SIMD architectures, which groups memory requests that are issued concurrently together within a warp. This reduces the workload on globally shared queues. Contrary, *ScatterAlloc* avoids single points of serialization and can thus provide memory requests in constant time, independent of the number of concurrent memory requests.

3. DESIGN GOALS

The general design goals for dynamic memory allocators are architecture-independent: **correctness**, **speed** and **little memory consumption**. To design an allocator for the GPU, we want to build on the know-how from the field of multi-core CPU memory allocation. In the following, we detail on the special requirements and distinct features that have to be considered for GPUs.

3.1 Correctness

A fundamental requirement for every deterministic algorithm is of course its correctness. A primary issue concerning the design of a GPU allocator is that it is virtually impossible to make any assumptions about scheduling on the GPU. This means that mostly lock-free algorithms have to be used to avoid deadlocks. Consequently, many well known CPU memory allocation methods drop out at this stage.

3.2 Speed

Keeping the number of clock cycles spent with allocation and deallocation of memory low, must be a major goal of allocators for the GPU. However, not only the time spent on the allocation and deallocation is important, but also how efficiently the allocated memory can be accessed.

Memory-access performance.

One important property of current GPUs is the fact that memory access can be extremely costly in relation to computations [14, 20]. On an Intel Core i7, data access takes about 4, 10, and 38 cycles, for data in L1, L2, and L3 cache respectively and about 100 cycles for accessing data in main memory [16]. On the Fermi architecture, access times are, according to our own measurements, about 18, 250, and 1000 cycles for data in L1 cache, L2 cache, and global shared memory respectively. Additionally the cache sizes on the GPU are about one tenth of the cache sizes on a CPU, which becomes even more severe considering that the L2 cache on a GPU has to serve hundreds of cores and thousands of threads. Therefore, *ScatterAlloc* keeps the number of data accesses low, while it can spend more time on complex computations.

Scalability.

A lock-free allocator usually relies on atomic operations to handle concurrent allocations of multiple threads as described in Section 2. The time it takes for such an operation to complete is proportional to the number of threads accessing the same data word in parallel. Although it is believed that a linear performance decrease is a good scalability for CPU-based allocators [2], it is not for GPUs where thousands of threads might execute concurrently. Our measurements show that, for multiple threads atomically accessing the same data word, the average access time increases with at least 100 cycles per thread. Consequently, atomic operations on the same data word are avoided whenever possible by *ScatterAlloc*. Ultimately, a perfect allocator should be unaffected by the number of threads concurrently allocating data and thus provide allocation and deallocation in constant time.

Diverging execution paths.

As the GPU is based on the SIMD model, blocks of threads (in our case CUDA-*warps*) can only execute in a coherent manner. This means, that if multiple threads within a *warp* concurrently allocate memory, the best performance will be reached, if they all execute the same code. Diverging branches are synchronized by the warp scheduler, forcing all threads to wait for the whole warp to finish, which can strongly reduce performance.

False sharing.

Another important issue with CPU-based allocators is *false sharing*. *False sharing* means that data accessed by different processors should not be placed in the same cache line if the data are not intended to be shared. Consequently, data accessed by one processor should be strung together to speed up the data access. Since the introduction of the *Fermi* architecture, GPUs also cache global memory accesses, making *false sharing* also an issue for efficient GPU programming. Because all threads executing on the same multiprocessor share the same cache, *false sharing* occurs between different multiprocessors only, while cache line sharing between different threads on the same multiprocessor can be intended.

Coalesced access.

Traditionally, data access on the GPU could only be performed efficiently if all threads of a block accessed the data words linearly according to their thread index, with no gaps in between, hence, in a coalesced manner. Although this is not necessary for current graphics cards anymore, data words accessed by adjacent threads can be read with a single load instruction, if they are placed close to each other in memory. If threads of the same block allocate data at the same point in time, it is likely that these data words will also be read or written concurrently. Thus, *ScatterAlloc* considers it important that data words allocated by threads of the same block at the same time are close to each other in memory.

3.3 Memory consumption

The lower the overall memory consumption of an allocator, the less data has to be accessed, the more data fits into the cache, and the more memory is free to be used by the system. Thus, a major design goal of dynamic memory allocators is to reduce the overall memory consumption. This includes memory fragmentation, which is a measure of the unusable regions between occupied memory, and furthermore the overhead of the data structures that are used to keep track of free and used regions. To be able to formalize our measures of fragmentation, we build on the assumption that memory is split up into regions. We define the set of all regions R as well as the functions $alloc : R \to \mathbb{N}$ and $size : R \to \mathbb{N}$. $alloc(r)$ maps a region to the size of the memory request it has been allocated for, or 0 if it is free. $size(r)$ gives the actual size of a region. Based on these functions, we can define the set of all allocated regions $A = \{r \in R | alloc(r) \neq 0\}$ and the set of all free regions $F = R \setminus A$.

Internal fragmentation.

Internal fragmentation occurs, if the size of the allocated memory region $size(r)$ is larger than the requested size $alloc(r)$. This may be necessary to meet alignment requirements of the processor, *e.g.*, for the Fermi architecture objects must be 16 byte aligned, or due to the internal structures of the allocator. Internal fragmentation may also vary with the allocation pattern of the application. We compute the internal fragmentation $F_{internal}$ as the average of relative wasted space among all allocated memory regions:

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

External fragmentation.

External fragmentation occurs, if the available free memory is divided into small chunks, which might be too small for direct use by the application. The amount of external fragmentation is strongly dominated by the allocator's strategy for finding free memory regions. Its value $F_{external}$ is commonly defined as the ratio of the largest free memory region to overall free memory:

$$F_{external} = 1 - \frac{\max_{f \in F} size(f)}{\sum_{r \in F} size(r)} \quad (2)$$

Consequently, an external fragmentation of 0 means that all available memory can be allocated in one big chunk.

Blowup.

Blowup occurs if the allocator's memory requirements disproportionally increase over time compared to the amount of allocated memory. The reason for this behavior can only be found by analyzing the internal structure of an allocator. In most cases, *blowup* is caused by the disability to reallocate previously freed memory. *Blowup* can dramatically increase memory usage for certain scenarios and is therefore considered in the design of *ScatterAlloc*.

4. BASELINE: A PARALLEL LIST-BASED ALLOCATOR

To obtain a baseline for the comparison with *ScatterAlloc*, we have implemented a traditional allocation scheme based on a *first-fit* algorithm on the GPU. Memory is organized in consecutive segments that each keep pointers to the previous and next segment. We refer to this structure as the *segment-list*. A memory segment can either be allocated or free. When a request is made for a new block of memory, the list of segments is traversed until a free segment of suitable size is found. If the size of the first fitting segment is larger than requested, the segment is split such that one segment then is of the appropriate size for the memory request and the other one contains the remaining memory. The first of the two parts is subsequently marked as allocated and returned to the caller.

To speed up the search process, we maintain a second list of free segments so that only those are considered during the search for a free block of memory. When a segment is allocated, it is removed from this *free-list* and if a split yields a free segment, this segment is inserted back into the list. To counter external fragmentation, the allocator attempts to merge segments back together with their immediate successor within the segment-list on deallocation.

For CPU allocators it is common to organize multiple heaps of smaller size to reduce allocation time. This is well demonstrated in numerous previous attempts [2, 4, 13, 15, 22, 24]. In an approach similar to *skiplists* [21], we introduce a partial ordering on the elements of the free-list, organizing it into sub-lists that contain blocks of certain sizes. A number of what we call bins each store entry points to these sublists. The way these bins are created and managed allows the implementation of various allocation strategies, including strategies similar to the use of multiple smaller heaps.

To build these data structures, management information has to be associated with every memory block. This information consists of a bit-field defining the current state of the block, as well as pointers to the previous and next blocks in both, the segment-list and the free-list. We store this information right before the actual memory block. Dealing with a highly parallel architecture, we want to support concurrent insertion and deletion of list elements. Thus, access to these pointers needs to be synchronized to avoid corruption.

Synchronization is also needed during the allocation process because multiple threads might try to allocate the same block of memory. If a thread identifies a suitable block of memory, it atomically tries to set the allocated flag. If this is successful, it is free to use the entire block or split it into two blocks.

With this functionality at hand, we have first created an allocator similar to dlmalloc [13] by inserting free memory blocks into the sublists according to their sizes. This enables a thread to directly jump to a first element that potentially satisfies its demands, instead of having to walk the entire free-list. As the individual sublists remain connected as a whole, a thread that has not been able to secure a block from its initial sublist will automatically run over to the next bin, splitting up a bigger block of memory.

For a second implementation we create a fixed number of bins and assign threads according to their execution locality to these bins. This creates a behavior similar to the Hoard system [2] and the 'Scalable lock-free dynamic memory allocation'-scheme [15]. We can assign all threads running on a single multi processor to the same bin, which has a positive impact on the number of cache hits and reduces congestion.

However, all these methods are not optimal for use on highly parallel architectures like GPUs, as already mentioned in Section 1 and confirmed by our results in Section 7. In the next section we therefore propose the novel design of *ScatterAlloc* that overcomes the inherent problems of the methods presented here.

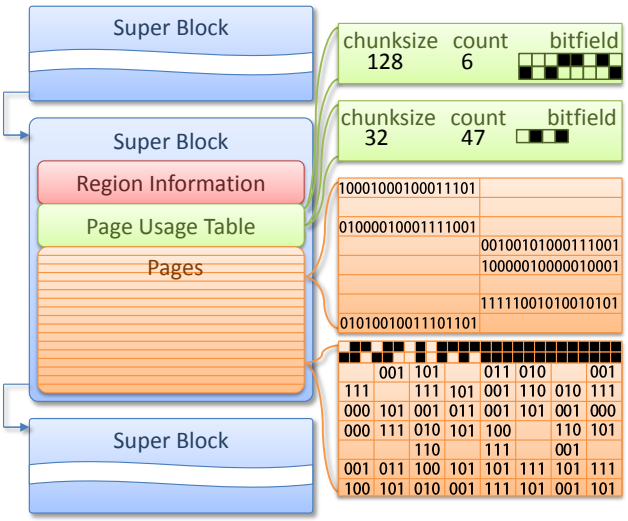

Figure 1: Overview of the data structures used in the *ScatterAlloc*: Memory is organized in super blocks which are collected in a list. Each super block holds a fixed number of equally sized pages, information about the usage of each page and meta information collected for regions of pages. The *usage table* captures the size according to which the page is split into equally sized chunks, the counter of used chunks and a bit field to identify the used chunks. To allow for splitting the page into more chunks than actually can be captured by the bit field in the usage table, an additional hierarchy of bits is placed on the beginning of the page.

5. ScatterAlloc

Traditional allocation strategies are too slow to be used where a large number of threads concurrently allocate memory. In an environment where hundreds of threads try to seize the first block of memory that fits their needs concurrently, collisions quickly become the major bottleneck of an allocator. To avoid these collisions, *ScatterAlloc* scatters allocation requests across fixed sized memory regions, trading fragmentation for allocation speed.

ScatterAlloc organizes its memory by splitting it into fixed sized *pages*. To keep track of the free memory within a page, we employ a *page usage table*. Pages are grouped together in *super blocks*, which form the biggest unit of memory for *ScatterAlloc*. To speed-up the search for allocable memory, we store meta data about the fill-level of different regions within each super block. This meta data is designed to be frequently read but rarely written in order to reduce congestion. For an overview of the data structures and their relations, see Figure 1.

5.1 Super blocks

On current NVIDIA GPUs, a fixed size memory pool is used to serve dynamic memory requests. The pool's size has to be set before a module is loaded into the context. Thus, it would currently be sufficient to design an allocator which can manage a memory region of a fixed size. As this behavior might change in future, we design our allocator in such a way that it can either be used on one big region of memory, or manage a variable number of memory blocks which can be found at arbitrary locations in memory. This enables *ScatterAlloc* to be applicable on future architectures, which might be able to increase the size of the heap during run-time. It is also possible to dynamically add memory to the pool manged by *ScatterAlloc* on current architectures. In case the allocator is almost out of memory, a host side *alloc* between two kernel launches can be used to allocate a new region of memory which is then passed to the allocator before the next kernel launch is executed. According to the common terms used for CPU allocators, we call these coherent regions of memory assigned to the allocator *super blocks*. We require all super blocks to be of equal size. If the allocator is configured to manage a single large region of memory, we split it up into multiple super blocks. To allow super blocks to be scattered in memory, we organize them in a singly linked list.

5.2 Pages

To facilitate parallel, collision-free memory allocation, we split every super block into fixed sized pages. This way, there is no need to search through lists when allocating memory because every thread can calculate the page offset from the size and address of the super block. To serve memory requests, pages are split into equally sized chunks. The chunksize on a page is set during the first access to the page. Once this split has occurred, it remains fixed until all chunks allocated on the page have been freed again. This strategy clearly favors the allocation of small and similarly sized chunks of memory. But this is actually the behavior we expect from a program using dynamic memory allocation in a highly parallel fashion: there will be multiple threads dealing with similar inputs, requesting similar amounts of memory. At the same time, these memory requests have to be small in relation to the entire available memory, because otherwise the system would run out of memory immediately.

To keep track of the chunks within a page, we facilitate a *page usage table*. Every entry in the page usage table consists of three values: the chunk size, the number of allocated chunks, and a bit-field. In this bit-field, each bit represents a single chunk of memory. A high bit means that the associated chunk of memory is in use, while a low bit indicates a free chunk. We require atomic operations and fast bit operators on this bit-field for *ScatterAlloc* to work efficiently. However, atomic operations are generally only supported for 32 or 64 bit words, with bitwise atomic operations only available on 32 bit words. Hence, the bit-field is 32 bit long in our current implementation. To support splitting a page into more than 32 chunks of memory and to handle smaller memory requests without a high amount of internal fragmentation, we introduce a second hierarchy level: On the page itself, we place up to 32 additional bit-fields, one linked to each bit in the page usage table. This way, we support splitting a page into up to $32^2 = 1024$ chunks.

5.3 Hashing

The key ability for a fast allocator on the GPU is to avoid collisions. If multiple threads try to allocate the same chunk of memory, performance can drop quickly. To avoid collisions and to quickly find suitable pages for allocation, we rely on hashing. While previous approaches use only the thread identifier for their hash function to determine a local heap, *e.g.* proposed by Larson *et al.* [11], our hash function fulfills additional requirements:

- When yielding a page, it seeks to split the page into chunks of the same size as the memory request. In this way, a free chunk on the page can be used and internal fragmentation will stay low.
- Threads running on the same multiprocessor aim for allocating memory side by side so that the cache utilization is higher.
- For our implementation in CUDA, threads within the same warp aspire to allocate their data adjoined as possible, as they will likely access data coherently.

To build the hash-function, we use the requested memory size and information about the location of the threads' execution (multiprocessor id). Using multiplicative hashing [10] we control the influence of the two factors, when determining a suitable page p:

$$p = \left(S_{requested} \cdot k_S + \mathrm{mp} \cdot k_{\mathrm{mp}}\right) \mod \mathrm{SP}, \tag{3}$$

where $S_{requested}$ represents the requested memory size, mp stands for the multiprocessor id, and SP denotes the number of pages within the super block. The factors k_S and k_{mp} can be used to distribute the access across the entire super block or to determine a local offset for the data. If we choose k_S to be a large, possibly a prime number, it is more likely that pages will be found which have been split with the same chunk size. Thus, internal fragmentation is kept low. A small value for k_S will cause little scattering of similarly sized memory requests. In combination with a large k_{mp}, data allocated by the same multiprocessor will be allocated close to each other. Thus, cache hits are more likely. However, in this case differently sized chunks will be placed on the same page, increasing internal fragmentation. The optimal choice depends on the usage pattern of the application.

In case the determined page is already full, we search for a free chunk in subsequent pages, which will introduce local clustering. This strategy results in better cache-utilization when searching for free pages, while the aforementioned considerations of either well matching chunk size or similar multiprocessor id also apply.

5.4 Meta Data

To speed up the search for free suitable chunks, we introduce a two level hierarchy of meta data. Meta data is read during each memory request, while it is only written occasionally. Our allocator always keeps a pointer to the currently active super block. Only if the super block reaches a certain fill level, or a memory request cannot be handled by the super block, the allocator proceeds to the next super block in the list. As a thread proceeds to the next super block, it updates the active super block pointer.

To quickly reject memory regions which are unlikely to contain suitable free chunks, we divide each super block into equally sized regions. For every region, we keep information about how many pages are full. If an allocation request fills up one page, it increases the counter for the region the page lies in. If a region is about to run out of free pages, memory requests are forwarded to the next region. As it takes some time for threads to report that pages are full, we already proceed to the next region when 90% of the pages are full. If a chunk of a full page is freed, the region counter is decreased, keeping the meta information accurate.

The available space within a region plays an important role in reactivating super blocks. Threads which free a chunk on a non-active super block randomly choose to set this super block as the active super block. The likelihood that this update is performed increases with decreasing fill-level. This helps to avoid memory blowup, as it allows for super blocks to be reused.

6. IMPLEMENTATION

To discuss the implementation details of our allocation strategy, we provide pseudo code of *ScatterAlloc*'s allocation and free methods. We require the target architecture to support atomic operations on global memory and efficient bit operators, as, *e.g.*, provided by the current CUDA C programming language.

6.1 Alloc

The pseudo code for *ScatterAlloc*'s allocation algorithm is split up in two functions: *alloc* and *tryUsePage*. Note that we omit the steps for placing more than 32 chunks on a single page, as they are very similar to placing only 32 chunks per page. The only difference is another hierarchical level, for which the bits in the page usage table are used to indicate that all associated 32 chunks of the second hierarchy are in use.

To allow placing smaller requests in bigger chunks, we introduce a multiplicative factor `max_frag` (see alloc line 3) to control how much space is allowed to be left empty between two adjacent chunks on a page. This factor is directly related to the maximum internal fragmentation.

While searching for a suitable page, three situations can occur:

- If the page's chunk size is either too small or too big to be used for the current request, we move on to the next page (alloc line 26).
- If the page's chunk size fits the current request, we try to use the page (alloc line 11-14).
- If the page is not yet in use, the page's chunk size is marked as zero. In this case, we use an atomic compare-and-swap with the goal to set the chunk size of the page according to our needs (alloc line 16-24). In case no other thread has set it before, the page can be used for allocation. If another thread set the chunk size in the meanwhile, it is still possible that the set chunk size fits the current request.

If *ScatterAlloc* identifies a page to be used for the current request, we first increase the fill level of the page to see if there is space for another chunk (tryUsePage line 2). In this way, we check if there is an available spot on the page concurrently with reserving a spot. After this step has been performed successfully, it is certain that the request will be served on that page. Without this counter, multiple threads might fight for the same spot. Note that this is the only point of serialization which depends on the number of threads trying to allocate memory concurrently. Nevertheless, not many threads will be directed to the same page as the hash-function will scatter their requests to multiple pages and full sections will be avoided due to the meta data structure.

After increasing the fill-level for the page, *ScatterAlloc* needs to determine which spot should be used. To do that efficiently, we use the bit-field of the page usage table (tryUsePage line 8-19). We start by estimating a free spot using the thread's lane id (thread index within its warp). This strategy is a good choice, because multiple threads of the same warp allocating equally sized data will be forwarded to the same page. In case all chunks are free, accessing them will result in coalesced memory accesses. If a chunk is already in use, we use the bit-field to quickly determine the next free spot as outlined in tryUsePage line 14-19. Essentially, we use bit-shifting in combination with CUDA's `__ffs` function to determine the closest free chunk on the page. Thus, we are able to find a free spot without the necessity to loop through the bit-field and mark the next free spot.

As long as our heuristic works well, there will be few or no collisions and a suitable page will be found quickly. In this case, an

Function alloc(Superblock,S_{req})

```
1  p ← hash(S_req, mp)
3  S_max ← S_req · max_frag
4  while tried not all regions do
5    region ← p/ regionsize
6    if region filllevel ≤ regionsize ·9/10 then
7      while p is in region do
8        page ← Superblock →p
9        S_chunk ← page →chunksize
11       if S_chunk ≥ S_req and S_chunk ≤ S_max then
12         loc ← tryUsePage(page, S_chunk)
14         if loc ≠ 0 then return loc
16       else if S_chunk = 0 then
           // page is free so try setting S_chunk
17         S_chunk ← atomCAS(page →chunksize, 0, S_req)
18         if S_chunk = 0 then
             // use the new page
19           loc ← tryUsePage(page, S_req)
20           if loc ≠ 0 then return loc
21         else if S_chunk ≥ S_req and S_chunk ≤ smax then
             // someone else acquired the page,
             // but we can also use it
22           loc ← tryUsePage(page, S_chunk)
24           if loc ≠ 0 then return loc
26       p ← p + 1
27   else
       // try next region instead
28     p ← (region +1)·regionsize;
29 return 0
```

Function tryUsePage(page,S_{chunk})

```
2  filllevel ← atomAdd(page →count, 1)
3  spots ← calcChunksOnPage(S_chunk)
4  if filllevel < spots then
5    if filllevel + 1 = spots then
6      atomAdd(page →region, 1)
8    spot ← laneId % spots
9    while true do
10     mask ← (1 « spot)
11     old ← atomOr(page →bitmask, mask)
       // if the spot is free use it
12     if old & mask = 0 then break
       // bit magic giving us the next free spot
14     mask = old » (spot + 1)
15     mask |= old « (spots - (spot + 1) )
16     mask &= (1 « spots) - 1
17     step ← __ffs( mask)
19     spot = (spot + step) % spots
20   return page →data + spot ·S_chunk
   // this page is full
21 atomSub(page →count, 1)
22 return 0
```

allocation request can be handled within a minimum of five global memory operations (determining the active super block, checking the region status, comparing the page's chunk size, increasing the fill-level, and marking the spot). If multiple threads try to allocate data at the same time, most of the information will already be in the cache and the request can be handled faster. We also benefit from the cache in finding a free section and searching through the pages, as this data occupies a continuous region of memory.

6.2 Free

Freeing a chunk of memory is simpler than allocating, as demonstrated by the pseudo code for *free*. Given that we already know which super block the chunk belongs to, a few address and offset computations (free line 2-7) are sufficient to determine the bit that has to be cleared to mark the chunk as free and the counter that has to be decremented to remove the reservation of the spot on the page.

In case a page has become completely free, *ScatterAlloc* resets the way the page has been split by setting the chunk size in the page usage table to zero. To avoid race conditions in this situation, we have to lock the page, so that no other thread will try to allocate memory on this page with an incorrect chunk size. We implement this lock by atomically setting the page's fill level counter to a maximum if it is still free (free line 12). Now, if a different thread tries to reserve a spot on the page, it will fail and we can safely reset the chunk size of the page. This functionality also justifies using this counter instead of using the bit-fields only, which would not be sufficient to resolve the concurrency issue if secondary hierarchies of bit-fields on a page are used.

```
Function free(SuperBlock,pointer)
2  p ← (pointer- Superblock →data)/pagesize
3  page ← Superblock →p
4  S_chunk ← page →chunksize
5  spot ← (pointer - page →data)/S_chunk
7  mask ← (1 « spot)
   // mark chunk free
8  atomAnd(page →bitmask, mask)
   // reduce counter
9  count ← atomSub(page →count, 1)
10 if count = 1 then
       // this page now got free, try 'locking' it
12     count = atomCAS(page →count, 0, pagesize)
13     if count = 0 then
14         page →chunksize ← 0
15         __threadfence()
           // 'unlock' it
16         atomSub(page →count, pagesize)
```

The provided pseudo code shows that two reads from global memory (data offset and chunk size) and two atomic operations on global memory are needed to free a chunk. The only source for congestion is formed by the two atomic operations. As the number of chunks on a page is limited, the bit-field operation will not cause a lot of congestion. The operation on the fill count may be subject to more congestion, as it is also atomically altered during the allocation process. Again, we can argue that the number of threads trying to request a chunk on a single page will still be low due to the use of a hash-function. Thus, the free operation is generally very efficient, as also shown in Section 7.

The only remaining question is how to determine the super block in which the chunk falls. In case of a single big allocator-managed block, a simple address calculation using one division is sufficient, because all super blocks lie next to each other in memory. When multiple separate super blocks should be used, a simple array with information about all available super blocks is all that is additionally required. While searching through this small array the L1 cache will be fully utilized and in an optimal case, the super block will be found about as fast as if we would only read a single value from global memory.

6.3 Large Data Requests

Up to now, we have described only requests that fit on a single page. In case a request is bigger than a page, we can serve this request by allocating multiple pages. As the used memory is strongly scattered within a super block, we forward requests that are bigger than the page size to super blocks that are deliberately reserved for this purpose. In this super block, we search for a sufficient number of consecutive free pages. Then we try to reserve these pages by setting all chunk size fields of each page's usage table to the requested size. This will prohibit any other thread from using the page in further memory requests. If in the meantime another thread tries to acquire one of the pages that we want to use, we reset all already reserved pages and restart the search. Note that such big data requests will not be made by a large number of threads, as the system would run out of memory very quickly.

7. EVALUATION

In this section we describe a set of tests to analyze the performance of dynamic memory allocators designed for highly parallel architectures such as GPUs. We build our tests in compliance with the design goals specified in Section 3. These benchmarks measure the allocation and free performance of the allocators, data access performance, and compute additional information such as fragmentation and overhead.

Operator performance.

The first interesting factor is the time it takes to fulfill an alloc or free request. This factor strongly depends on the internal state of the allocator (the allocation history and assigned memory), the number of threads being executed, the number of threads concurrently allocating/freeing data and the size of the requested memory.

To construct a realistic and at the same time challenging scenario, we test the allocators performance across multiple kernel calls. A thread can either allocate memory or free memory that it has allocated before. The probability that it does so is given by the test parameters p_{alloc} and p_{free}. After a couple of kernel calls, the ratio between allocated and free memory will approach a constant. Subsequently to this initial warm-up phase, we measure the average number of cycles required for a single alloc or free. Furthermore, the variance in cycles between different allocation requests is of interest. The coefficient of variation can be seen as a characteristic value of the allocator's homogeneity when it comes to diverging execution paths. To simulate different load characteristics, we increase the number of threads until the device's capacity is reached or the allocator runs out of memory. To capture the influence of the requested memory sizes, they are drawn from an uniform distribution.

Data access performance.

Besides the time for allocating/freeing memory, the time it takes to access the allocated memory influences the overall performance

of a program as well. It strongly depends on the access pattern and cache-utilization. We extend the previously designed test to also measure how long it takes each thread to access the memory it allocated.

Meta Information.

Not only the raw performance measures give information about an allocator's performance. Internal fragmentation, external fragmentation and memory overhead of the required data structures are also important. Using the previously described test, we can measure these factors to analyze the allocator's performance in different situations according to their definition as given in Section 3.

8. RESULTS

To compare the performance of *ScatterAlloc* with state-of-the-art CPU allocators on current GPU hardware, we have implemented two list based allocators as described in Section 4 using CUDA. Thus, we can also compare their performance directly with the allocator provided with the CUDA toolkit 4.0 and *XMalloc* [8]. Our test-system is equipped with an NVIDIA Quadro 6000 with 6GB of graphics memory, of which we assign 16MB to each of the allocators. The *size list* allocator uses ten bins for speeding up the search for differently sized data blocks. The *mp lists* allocator sorts free memory blocks in ten bins per multiprocessor, which makes for an overall bin-count of 140. For our allocator, we use two randomly selected prime numbers $k_S = 38183$ and $k_{mp} = 17497$ as the respective parameters of our hash-function, a pagesize of 4kB, and a super block size of 8 MB. According to our tests, the performance of *ScatterAlloc* is not sensitive to the choice of k_S and k_{mp}, as long as they are large prime numbers.

Figure 3 shows the performance of the alloc and free operators for an increasing number of threads, with $p_{alloc} = p_{free} = 0.75$. As the purpose of this test is to assess the impact of concurrent allocation, we stopped the tests after exceeding the point where the GPU was fully utilized (at about 15000 threads). One can clearly see, that the performance of the list based allocators, *XMalloc* without SIMD optimization, and the CUDA toolkit allocator strongly decreases with an increasing number of threads. *XMalloc* with SIMD optimization can reduce the number of memory request served via the global queues and can avoid a performance decrease until approximately 2000 threads are concurrently allocating memory. The performance of *ScatterAlloc* remains almost constant as the thread count increases. At full utilization of the GPU, memory allocation using *ScatterAlloc* is about 100 times faster than the CUDA toolkit allocator and up to 10 times faster than *XMalloc* with SIMD optimization. Our allocator's performance varies with the requested memory size, due to its page based strategy. If the queues in *XMalloc* become empty, allocation takes a lot longer, which causes the variation in *XMalloc*'s performance. The other allocators show little variance for their allocation time. Comparing the two list based allocators, shows that distributing memory requests of different multiprocessors to different memory-locations yields an increase in performance of about 1000%.

To assess *ScatterAlloc*'s memory consumption characteristics, we compare it to list-based allocators. They work similar to allocators for CPUs, which have been tuned for low memory consumption and fragmentation. Figure 4 shows the internal and external fragmentation measured for the list based allocators and *ScatterAlloc* for different distributions of allocation request sizes. We set $p_{alloc} = p_{free} = 0.6$, the mean of the size-distribution to 512 bytes, and varied the extent of the distribution from zero to 896 bytes. The performance of the tested allocators was constant for all distributions and equaled the measurements reported in Figure 3. Internal

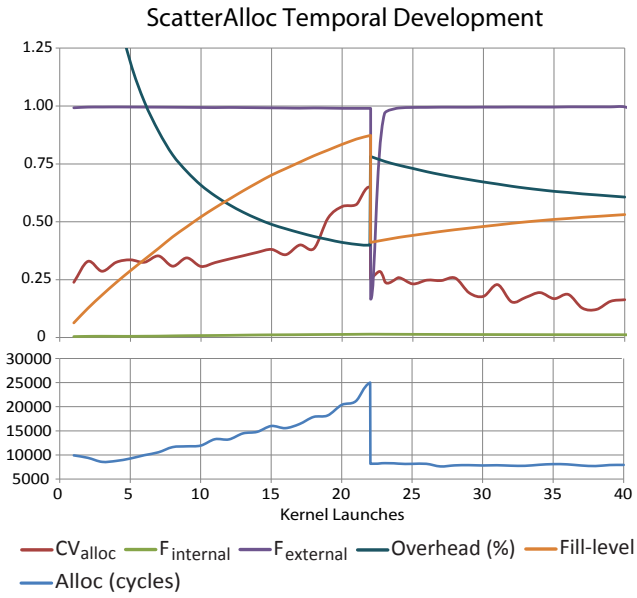

Figure 2: Temporal development of *ScatterAlloc*. 16384 threads keep requesting memory chunks between 128 and 160 bytes. As the first super block fills up, the time needed for finding a free chunk starts to vary (CV) and thus increases. After a fill level of about 80% is reached, the next super block is opened up and performance recovers. Internal fragmentation is very low, while the external fragmentation is linked to allocation speed. The relative overhead of *ScatterAlloc* stays below 1% after a reasonable fill level has been reached.

fragmentation was very low for all allocators, indicating that *ScatterAlloc* finds good target candidates most of the time. *ScatterAlloc* as well as the mp lists allocator trade external fragmentation for increased allocation performance. Thus, it is not surprising that the external fragmentation of these allocators is very high. *ScatterAlloc* produces more cache hits than the list based allocators, resulting in faster access. As the latency for the L1 cache is about 18 cycles and about 250 cycles for the L2 cache, the results indicate that all three allocators deliver memory in a cache friendly way.

Figure 2 shows the temporal development of *ScatterAlloc* for 16384 threads, two super blocks of 8MB, uniformly distributed allocation requests between 128 and 160 bytes, and with $p_{alloc} = 0.05$ and $p_{free} = 0$. While the first super block fills up, the time needed for finding a free spot increases from 10000 to 25000 cycles. The coefficient of variation (CV) for the allocation time also increases, which indicates that for some threads it becomes more difficult to find a free spot. When there is hardly any space left for allocation, the external fragmentation drops, and the time spent on a single allocation request reaches its maximum. If the allocation request can not be served by the first super block, the next super block is opened up and the allocation time drops below 10000 cycles again. The relative overhead of the allocator for a reasonable fill level is below 1%. As the overhead is constant per super block, the relative overhead decreases with increasing fill-level.

For memory requests larger than the used page size, *ScatterAlloc* also shows a linear performance decrease with increasing number of threads. For a single thread allocating two pages takes approximately 8000 cycles. For every additional thread allocating data, the average performance decreases with approximately 3000 cycles.

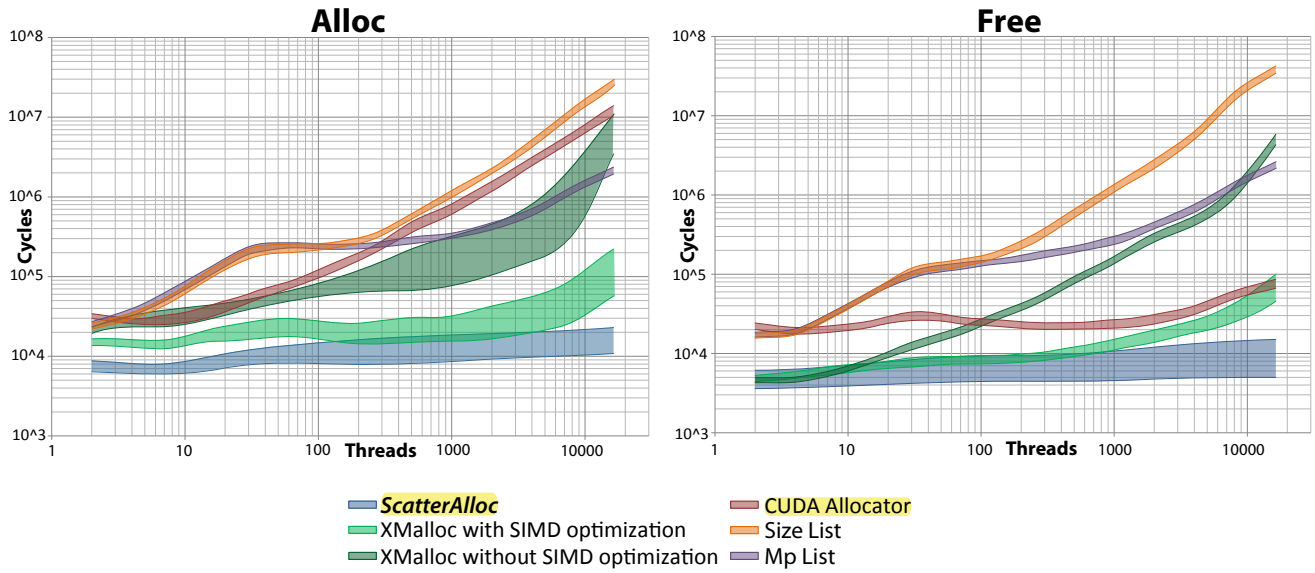

Figure 3: Number of clock cycles needed to serve a single alloc or free for 16, 32, 64, and 128 bytes as the number of threads concurrently allocating memory increases. The performance of the list-based allocators, *XMalloc* without SIMD optimization, and the CUDA toolkit allocator strongly decrease with increasing thread count. *XMalloc*'s SIMD optimization can postpone the performance decrease until about 2000 threads are active. Contrary, the performance of *ScatterAlloc* is almost independent of the thread count, outperforming the other allocators over the entire test interval. The CUDA toolkit allocator failed in the test runs with 128 byte allocation size. Thus, this data had to be omitted from above measurements.

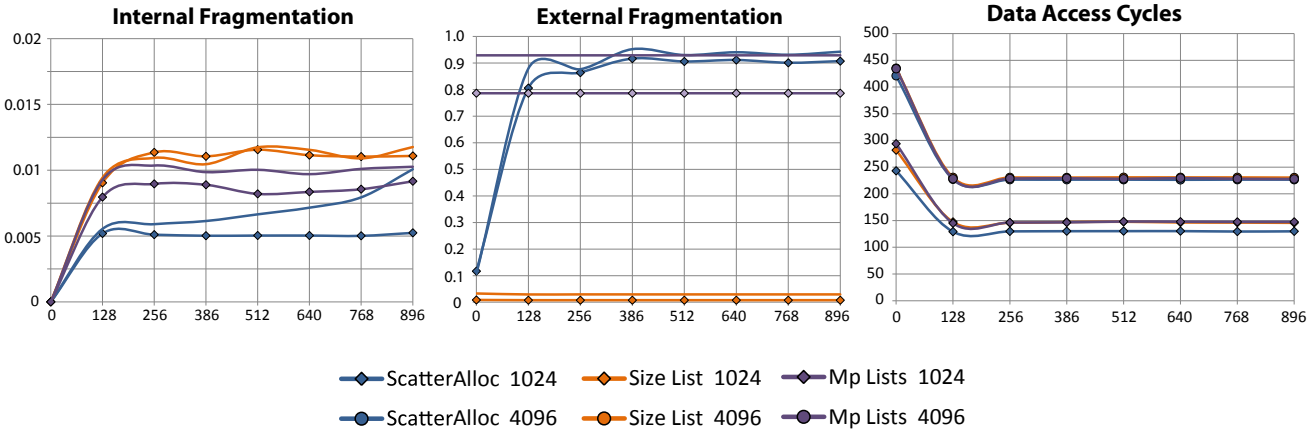

Figure 4: Internal and external fragmentation as well as data-access performance for the list-based allocators and *ScatterAlloc* for increasing variation in allocation request size (in bytes), for 1024 and 4096 threads. The mp lists allocator and *ScatterAlloc* both trade fragmentation for speed. Although *ScatterAlloc* uses a page-based memory model, its internal fragmentation is comparable to the list-based allocators. Comparing the achieved access times to the latencies of the L1 cache (18 cycles) and the L2 cache (250 cycles) indicates that all allocators deliver memory in a cache friendly way, whereas *ScatterAlloc* slightly outperforms the other allocators for a lower number of threads.

9. CONCLUSION

We have discussed several necessary design goals which have to be met by efficient dynamic memory allocation schemes for massively parallel computing device architectures. In this paper, we have focused on GPUs, which currently provide the highest degree of parallelism on a single device. The special needs of thousands of GPU-threads, which may concurrently need to allocate memory, require the implementation of new dynamic memory allocations schemes. We described how established CPU-based allocation methods can efficiently be tailored to the GPU. However, we also show that these methods are still too slow to meet the demands of massively parallel programs. Hence, we have implemented our own allocator, *ScatterAlloc*, which turns out to be more reliable and about 100 times faster than the built-in CUDA memory allocation function and up to 10 times faster than the to our work related *XMalloc*. *ScatterAlloc* is open source and can be downloaded from our project web page: `www.icg.tugraz.at/project/mvp`.

10. ACKNOWLEDGMENTS

We would like to thank Christopher Rodrigues from the University of Illinois for providing the original implementation of *XMalloc*. This research was funded by the Austrian Science Fund (FWF): P23329.

11. REFERENCES

[1] J. H. Ahn. *Memory and control organizations of stream processors*. PhD thesis, Stanford University, Stanford, CA, USA, 2007. AAI3253463.

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, Nov. 2000.

[3] D. Cederman and P. Tsigas. On sorting and load balancing on gpus. *SIGARCH Comput. Archit. News*, 36:11–18, June 2009.

[4] D. Dice and A. Garthwaite. Mostly lock-free malloc. In *ISMM '02*, pages 163–174, New York, NY, USA, 2002. ACM.

[5] W. Gloger. ptmalloc. http://www.malloc.de/en/.

[6] D. Häggander and L. Lundberg. Optimizing dynamic memory management in a multithreaded application executing on a multiprocessor. In *ICPP '98*, pages 262–269, Washington, DC, USA, 1998. IEEE.

[7] M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison Wesley, Aug. 2007.

[8] X. Huang, C. Rodrigues, S. Jones, I. Buck, and W. mei Hwu. XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines. In *Computer and Information Technology (CIT), 2010 IEEE*, pages 1134 –1139, July 2010.

[9] R. L. Hudson, B. Saha, A. Adl-Tabatabai, and B. C. Hertzberg. McRT-Malloc: a scalable transactional memory allocator. In *ISMM '06*, pages 74–83, New York, NY, USA, 2006. ACM.

[10] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[11] P.-A. Larson and M. Krishnan. Memory allocation for long-running server applications. *SIGPLAN Not.*, 34:176–185, October 1998.

[12] C. Lauterbach, Q. Mo, and D. Manocha. gproximity: Hierarchical gpu-based operations for collision and distance queries. *Comput. Graph. Forum*, 29(2):419–428, 2010.

[13] D. Lea. A memory allocator. unix/Mail, http://gee.cs.oswego.edu/dl/html/malloc.html, 1996.

[14] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, Apr. 2008.

[15] M. M. Michael. Scalable lock-free dynamic memory allocation. *SIGPLAN Not.*, 39(6):35–46, June 2004.

[16] D. Molka, D. Hackenberg, R. Schone, and M. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09.*, pages 261 –270, 2009.

[17] D. M. Nicol. Inflated speedups in parallel simulations via malloc(). *International Journal on Simulation*, 2:413–426, 1992.

[18] Nvidia. *Fermi Compute Architecture Whitepaper*, 2009.

[19] Nvidia. *NVIDIA CUDA C Programming Guide*, June 2011.

[20] NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara 95050, USA. *CUDA C Best Practices Guide*, 4.0 edition, May 2011.

[21] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33:668–676, June 1990.

[22] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *ISMM '06*, pages 84–94, New York, NY, USA, 2006. ACM.

[23] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH '08*, pages 18:1–18:15, New York, NY, USA, 2008.

[24] S. Seo, J. Kim, and J. Lee. Sfmalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores. In *PACT '11*, pages 253–263, Washington, DC, USA, 2011. IEEE Computer Society.

[25] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*, 1995.

[26] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27:126:1–126:11, December 2008.