Fast Dynamic Memory Allocator for Massively Parallel Architectures

Sven Widmer
Graduate School
Computational Engineering
TU Darmstadt

Dominik Wodniok
Graduate School
Computational Engineering
TU Darmstadt

Nicolas Weber
TU Darmstadt

Michael Goesele
Graduate School
Computational Engineering
TU Darmstadt

ABSTRACT

Dynamic memory allocation in massively parallel systems often suffers from drastic performance decreases due to the required global synchronization. This is especially true when many allocation or deallocation requests occur in parallel. We propose a method to alleviate this problem by making use of the SIMD parallelism found in most current massively parallel hardware. More specifically, we propose a hybrid dynamic memory allocator operating at the SIMD parallel warp level. Using additional constraints that can be fulfilled for a large class of practically relevant algorithms and hardware systems, we are able to significantly speed-up the dynamic allocation. We present and evaluate a prototypical implementation for modern CUDA-enabled graphics cards, achieving an overall speedup of up to several orders of magnitude.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: —Language Constructs and Features—*dynamic storage management*; D.4.2 [**Operating Systems**]: Storage Management—*Allocation / deallocation strategies*

Keywords

parallel computing, dynamic memory allocation, GPGPU

1. INTRODUCTION

Dynamic memory allocation is one of the most basic features programmers use today. It enables memory allocation at runtime and is especially useful, if the amount of memory needed is not known ahead of time. Modern operating systems provide therefore easy to use interfaces to allocate and free memory arbitrarily. Unfortunately, these approaches do not generalize directly with good performance on massively parallel architectures such as current graphics processing units (GPUs) or even many-core systems. The key problem is hereby that bookkeeping during naïve allocation and deallocation requires a form of global synchronization. This is a severe performance bottleneck when systems become more and more parallel.

This effect can, e.g., be observed in practice when using the C functions `malloc(size_t)` and `free(void*)` that were recently included into NVIDIA's CUDA framework [11] to allocate memory dynamically at runtime. Several approaches have therefore been developed to build a dynamic memory allocator capable of working in a massively parallel environment, where traditional approaches do not work, including XMalloc [5] and ScatterAlloc [12]. Although these implementations show better results than the built-in CUDA allocator, their application still results in a noticeable slowdown.

Our key observation is that massively parallel architectures typically operate in a SIMD fashion where a single instruction is physically executed in parallel. In CUDA, this corresponds to the concept of a warp. Memory allocation and deallocation should take this into account and will ideally yield a significant speedup when operating in this granularity. We additionally propose multiple constraints and assumptions which are fulfilled in many practically relevant algorithms and hardware systems that yield a different and much faster implementation. In particular, we assume that a systemwide default memory allocator is available. Further, we expect the application to free memory not arbitrarily but free all memory at certain points during the execution.

These assumptions fulfill the SIMD parallel programming scheme and can be applied to various algorithms, in particular algorithms with an unpredictable transient or output data size such as Monte Carlo-based simulation techniques, graph layout algorithms, or adaptive FEM simulations. All these algorithms can be implemented without dynamic memory allocation by interrupting the GPU computations at critical points and allocating additional needed memory. The introduced global synchronization can be at

GPGPU-6, March 16, 2013, Houston, TX, USA.

least partially reduced if not eliminated by use of dynamic memory allocation.

Our contributions are as follows:

1. an improved memory allocator for massively parallel architectures with a wide SIMD width such as Nvidia's CUDA and Intel's Xeon Phi. It drastically increases performance and works for a wide variety of applications.

2. a comparison of the proposed allocator with the standard CUDA malloc and ScatterAlloc.

The remainder of this paper is organized as follows: We first discuss related work on dynamic memory allocation for CPU and massively parallel architectures in Section 2. We then introduce the concepts behind our fast GPU memory allocator (Section 3) and describe a prototypical implementation (Section 4). Next we discuss the limitations of our approach and evaluate our results (Section 5). Finally we conclude and give an outlook on future work (Section 6).

2. RELATED WORK

Dynamic memory allocation is nowadays an ubiquitous operation. Therefore most programming languages provide some mechanism to allocate memory at runtime. The complexity of such operations is transparent for most programmers. They are in most cases not aware of the implications the operations have on the overall runtime of an application. This is specially the case for multi-processors and in particular multi-core CPU systems.

Many different memory allocators and algorithms for memory management where proposed over time. An introduction and basic overview is given by Knuth [8] and Tanenbaum [13]. Wilson et al. [14] have evaluated and compared the most common algorithms regarding the overall memory consumption.

Over the past years a lot of dynamic memory allocators for multi-processor and multi-core systems were proposed. Gloger [3] extended the well known and widely used dlmalloc [9] to support multi-thread environments. Hoard by Berger et al. [1] uses per processor heaps in addition to a global heap to increase scalability. Dice and Garthwaite [2] as well as Michael [10] introduced memory allocators based on lock-free data structures.

Modern parallel architectures have a wide variety of active hardware threads. They range from multi-core systems with up to 16 cores over many-core systems to massively parallel architectures such as GPUs which can execute thousands of threads concurrently. With the increasing number of threads running concurrently the sychronization overhead increases and becomes a severe bottleneck. As a result the scalability and in particular the SIMD scalability decreases.

In this work we, will focus on dynamic memory allocation for massivly parallel architectures with a wide SIMD width such as GPUs or Intel's Xeon Phi. Our main goal is hereby to increase the SIMD scalability.

2.1 Dynamic memory allocation on CPU

With the introduction of multi-processor and multi-core systems the classic memory allocation algorithm became a severe bottelneck. To increase the scalability of an allocator on a multi-processor system Häggander and Lundberg [4] proposed two optimizations. A parallel heap and memory pools for commonly used object types were implemented to gain a significant speedup. For every processor a heap area is created. A thread can allocate memory using this area of the processor it is executed on. If the heap area is occupied by a different thread one can try to lock a heap area of another processor. A similar approach, the Hoard allocator, was proposed by Berger et al. [1]. They augment a global heap with a per-processor heap that every thread may access. Hoard caches a limited number of superblocks (a chunk of memory) per thread. To keep fragmentation at a minimum unused superblocks can be migrated into the global heap and used by other processors.

Most dynamic memory allocators rely on atomic operations or even require mutual exclusion locks to handle their critical section and keep shared data structures consistent. This can have a significant performance impact and reduced scalability with increasing number of cores per processor. To reduce mutual exclusion locks lock-free data structures which build on the atomic Compare-and-Swap (CAS) or the Load-Linked and Store-Conditional (LL/SC) operation are supported by almost all current CPU architectures. In this context Michael [10] presents a completely lock-free memory allocator.

Hudson et al. [6] presented McRT-malloc, a scalable non-blocking transaction-aware memory allocator that is tightly integrated with a software transactional memory system. It avoids expensive CAS operations by accessing only thread-local data and increases scalability even further.

2.2 Dynamic memory allocation on massively parallel architectures

Publications dealing with dynamic memory allocation for GPGPU applications are scarce. Huang et al. [5] introduce the problems that have to be faced when building a memory allocator for massively parallel architectures, identifying the need to synchronize access to header data as the main issue. This synchronization serializes execution and therefore decreases the performance gain from parallel architectures.

XMalloc uses a similar approach as the Hoard allocator [1] by introducing superblocks and using the atomic CAS operation from Michael [10] to reduce synchronization overhead. To parallelize memory allocation, all threads are divided in smaller groups. Each group maintains their own superblocks. This allows groups to work independently from each other and to only access the global memory management when allocating a new superblock. Each superblock can be divided into several smaller blocks and distributed among the threads in the group.

ScatterAlloc by Steinberger et al. [12] expands XMalloc [5] by introducing a new approach to further reduce simultaneous access from different threads to the same memory region. Their implementation does not search linearly for a free memory slot, but instead scatters the memory access. This reduces the concurrent access to the same memory region and speeds up the allocation process.

In contrast to XMalloc and ScatterAlloc we introduce a voting algorithm to determine a single worker thread, reducing the amount of concurrent access to the critical section and increasing SIMD scalability. We propose assumptions that minimize the algorithmic complexity and suggest an allocation principle based on superblocks that increases the overall performance.

3. DESIGN

Various modern many-core architectures (e.g., GPUs or accelerator cards) are executing a group of threads in a SIMD style fashion, each thread corresponding to a SIMD lane. All those threads must execute the same instruction to minimize divergence and achieve the best performance. A dynamic memory allocator for those systems must scale when thousands of threads allocate different chunks of memory at the same time. In the following, we call a group of SIMD lanes of one streaming processor core a warp, similar to the CUDA terminology. An important consequence of the SIMD nature is that threads in a warp are implicitly synchronized.

Allocation algorithms such as Hoard [1] are optimized for multi-threading environments. They do not scale with increasing numbers of warps. Allocators proposed for many-core architectures (e.g. XMalloc [5]) are too general since they are based on the assumption that all threads are independent and not executed in SIMD style.

The main design goal for our allocator was to increase the SIMD scalability for small, frequent memory allocations and therefore endeavor to reduce the branch divergence. To achieve this, we rely on the following three assumptions.

1. A system wide default memory allocator exists and works fast, as long as there are only few simultaneous requests.

2. There is no need to free single chunks of memory during the execution. It is sufficient that the complete allocated memory of a group of threads can be freed at a certain point during the execution.

3. Most memory requests are smaller than some threshold.

ScatterAlloc [12] and XMalloc [5] are using superblocks, a chunk of memory that is allocated for a group of threads. They divide this superblock into several smaller memory chunks that are only accessed by this group. This reduces the number of global memory allocation requests and there is no need for global synchronization when manipulating a superblock. In our approach one superblock is shared by all threads in a warp. To reduce the simultaneous memory requests a voting is performed that determines a so called worker thread. This thread does all the work for his group. Thereby we can reduce the invocations up to SIMD width times.

Our second assumption makes it obsolete to have any header data for the superblock except for one pointer register, which points to the next unoccupied chunk inside the superblock. With this simplification, the time needed to allocate memory inside a superblock is reduced significantly. We further reduce the synchronization and memory overhead introduced by a general `free(void*)` method.

The last assumption ensures that the default allocator is used as little as possible. To guarantee not only good performance in allocating memory but also efficient cache use, we aggregate all memory requests inside a warp.

3.1 Data Layout

Similar to the parallel heap used by Hoard [1] as well as Häggander and Lundberg [4] we create a heap per warp accessible for all threads in the corresponding warp. The so

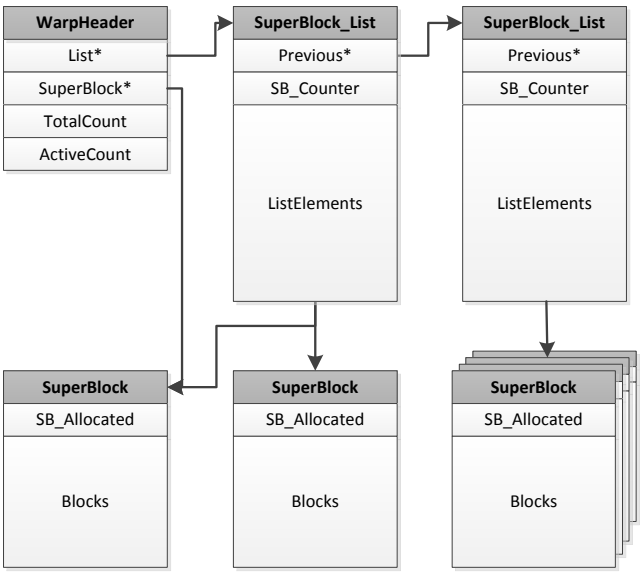

Figure 1: Overview of the data layout and organization of the three data structures used by our allocator.

called **WarpHeader** organizes all memory requests inside a group of SIMD lanes. Figure 1 shows the data layout and how the objects are organized. Each header contains a pointer to the current **SuperBlock** and a pointer to a list (**SuperBlock_List**) that stores all pointers to superblocks that have been allocated using the default memory allocator. The size of the list is fixed. If it is full, it is replaced by a new empty list and the old list is registered in the new list, so that the reference is not lost. **SB_Counter** denotes the number of allocated elements in the list. Besides its memory allocation region, a superblock also stores the amount of allocated memory in **SB_Allocated**.

Two additional variables are stored inside the warpheader for later use. The **TotalCount** describes the number of threads inside a warp which use dynamic memory allocation. The second **ActiveCount**, contains the number of threads that have not finished execution.

3.2 Initialization

To initialize the dynamic memory allocator, every thread that needs the ability to request memory dynamically, has to obtain a warpheader. At first all threads inside a warp have to determine how many threads in a warp require the warpheader. This can be realized by a voting function. After that a worker thread has to be declared by using the position of the most significant set bit of the voting mask as the ID of the thread. The worker thread allocates the warpheader using the default memory allocator and distributes the pointer to the other threads. This worker thread also allocates the current superblock and registers it in the list. The algorithm is illustrated in Figure 2.

3.3 Allocation

The allocation process is divided into two phases (see Figure 3). In the first phase, the required memory amount of

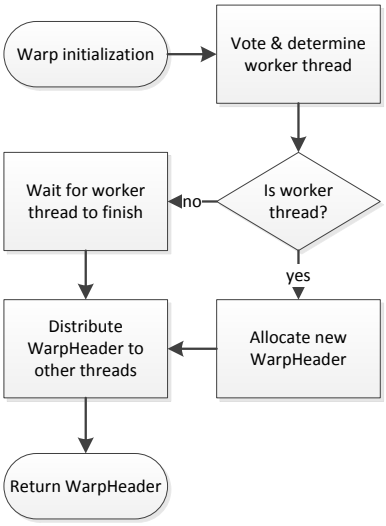

Figure 2: Warp execution flow for initialization of the dynamic memory allocator.

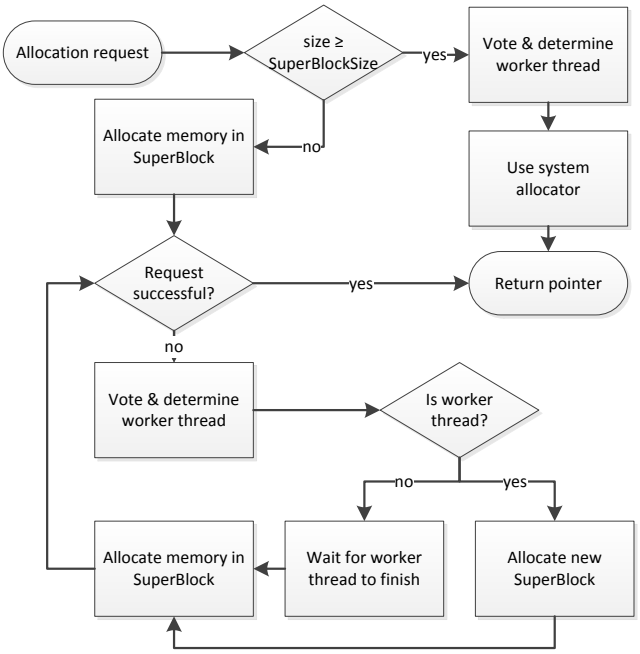

Figure 3: Warp execution flow for a memory allocation request.

each requesting thread is mapped to the next multiple of a minimum allocation block size. The default minimum allocation block size is 16 bytes, but it can be adjusted to fit the application needs. Memory can only be allocated in blocks, therefore allocating 17 bytes would result in a 32 byte allocation. This guarantees correct alignment inside memory for better cache reads and writes.

In the second phase, it is checked if the total requested memory size of all requesting threads is smaller than the maximum superblock size. If this is the case, the threads try to allocate memory in the associated superblock. All threads that have not been able to allocate memory in the superblock again perform a voting to decide on a worker thread, which allocates a new superblock and registers it in the superblock list. The remaining threads allocate memory in the new superblock.

If the total requested memory size exceeds the maximum superblock size the whole request is served using the default allocator. The returned pointer is registered in the superblock list, so that the developer does not need to know whether the request was handled by a superblock or the default allocator.

As mentioned before, there is no need for any header data in a superblock except for a pointer register, which points to the last used block. To enable coalesced memory access, all requests are mapped to a contiguous memory region. All active threads calculate the sum of requested memory for all other threads up to their own thread ID. This can be done by prefix sum or a simple iteration. The sum is used as an offset to calculate the thread's own block position in the superblock. The active thread with the highest ID has all information needed to update the allocation status information of the superblock.

3.4 Garbage Collection

The proposed memory allocator keeps track of allocated memory using a list. To free these allocations we use three strategies:

1. **Clean all dynamically allocated memory:** A function traverses all lists and frees all stored memory pointers. After this, a new empty superblock will be allocated.

2. **Shutdown dynamic memory allocation:** All allocated memory including the warpheader is deallocated. This strategy is meant to be invoked at the end of the kernel, so that all memory is freed correctly and no memory leaks occur. After this function has been executed, it is no longer possible to allocate memory.

3. **Make allocated memory available to other kernels after kernel termination:** In contrast to clean up the complete warpheader at the end of kernel execution a pointer to the warpheader is returned. This allows us to use allocated memory over several kernel calls but it is still required to eventually invoke the previous strategy.

In each strategy the calling threads atomically decrement **ActiveCount**. The thread that reduces the count to zero executes the respective strategy. For the first strategy it additionally resets **ActiveCount** to **TotalCount**. This follows from the assumption that all threads with a warpheader keep the ability to dynamically allocate memory.

3.5 Constraints and Limitations

Our proposed allocator design implies a few constraints which we summarize here. These constraints must be fulfilled to guarantee fast and correct operation.

- All constraints of the default allocator are inherited and are still applicable.
- Threads that have requested a WarpHeader eventually must call one of the clean up functions. Otherwise the memory will not be freed as described in Section 3.4.

- The used hardware must provide a voting function for an efficient implementation.

4. IMPLEMENTATION

We used Nvidia's CUDA Version 5.0 [11] to implement our proposed allocator (Fast Dynamic GPU Memory Allocator – **FDGMalloc**). The design is kept very generic. This allows the system to be used with most current many-core architectures (e.g., Intel Xeon Phi [7]) if the architecture supports a warp voting function (see Section 3).

4.1 Allocator on GPU using CUDA

As Figures 4 and 5 illustrate, the CUDA toolkit built-in memory allocation function (**CUDAMalloc**) is fast for large and few simultaneous allocation requests. Therefore, it is used for the allocation of new superblocks. The bottleneck for CUDAMalloc (as well as for most other allocators) is the SIMD scalability. We reduce the amount of concurrent requests and therefore increase the scalability.

To determine a worker thread we use the CUDA voting function *__ballot*, which essentially returns the lane mask for all participating threads when called with a predicate not equal to zero. The corresponding bit of non-participating threads is automatically set to zero. Afterwards we us *bfind* to find the most significant set bit and declare the corresponding thread as the worker thread.

To create a warpheader all threads in a warp use the CUDA voting function *__ballot* to determine how many threads in a warp require the warpheader. The worker thread allocates the warpheader using CUDAMalloc and distributes the pointer to the other threads.

The distribution of the pointer to the warpheader as well as later the pointer to a chunk of memory can be realized in two different ways. For compute capability 2.0 shared memory is used to distribute the pointer to the other threads. If compute capability 3.0 or higher is available, the pointer exchange between threads within a warp is realized using the function *__shfl*, removing the need for shared memory. Performance analysis has shown, that there is no difference in execution time by using shared memory or the *__shfl* function.

As described in Section 3.3, all allocation requests that are smaller then a certain threshold are served by a superblock without any atomic operations involved. In this case we reduce the number of global memory allocation requests and there is no need for global synchronization or thread serialization.

4.2 Usage

Listing 1 shows a simple example of how to use the allocator. All threads request a warpheader, allocate some memory, and free all memory at the end of the kernel. The example demonstrates usage of the *tidyUp()* function, which implements the first strategy for garbage collection. Each thread allocates some memory every time the loop is executed. memory is freed at the end of every loop cycle. After the execution of the warp the warpheader and all other header data is freed.

To prevent memory leaks, the function *end()* has to be executed at the end of the kernel. Otherwise at least one list, one superblock, and the warpheader per warp will not be freed. The example in Listing 2 shows a misusage of our allocator. Not all threads that allocate memory have also requested the warpheader. Further not all threads that requested the warpheader also exectue the *end()* function. As a consequence, the warpheader will not be freed.

```
void __global__ kernel(void) {
  Warp* warp = Warp::start();

  while(condition) {
    void* ptr = warp->alloc(size);

    /* ... some code ... */

    warp->tidyUp();
  }

  warp->end();
}
```

Listing 1: Simple usage example with first strategy for garbage collection (Section 3.4).

```
void __global__ kernel(void) {
  Warp* warp = 0;

  if(threadIdx.x % 5 == 0)
    warp = Warp::start();

  /* ... some code ... */

  void* ptr = warp->alloc(size);

  /* ... some code ... */

  if(threadIdx.x == 5)
    warp->end();
}
```

Listing 2: Example of a misusage: not all threads request a warpheader but try to allocate memory.

5. EVALUATION

All experiments were performed on a PC running Windows 7 64-bit version and the latest Nvidia driver (version 306.97). The system is equipped with an Intel Core i7 920, 12 gigabytes of RAM, an Nvidia Geforce GTX 480 (primary device), and a GTX 680 with 2 gigabytes of RAM (headless device). All tests were performed on the Geforce GTX 680.

5.1 Different allocation sizes

First, we compare the proposed allocator with the default CUDA allocator (CUDAMalloc) and ScatterAlloc [12].

In the following test scenario X threads are created which allocate N times S bytes of memory. At the end all memory is freed. For FDGMalloc this is done by invoking the *end()* method. In the case of CUDAMalloc and ScatterAlloc it has to be done by hand. We measure the runtime performance by allocating memory chunks of different sizes. Therefore the parameters for the test scenario have been set to $X = \{64\}$, $N = \{16\}$ and $S = \{16, 32, 64, \ldots, 8160, 8176, 8192\}$. The size of a superblock is 32 kilobytes and the minimum allocation block size is 16 bytes. The list of a warpheader contains 126 entries to manage superblocks before allocating a larger list. To analyze the impact of FDGMalloc's alloca-

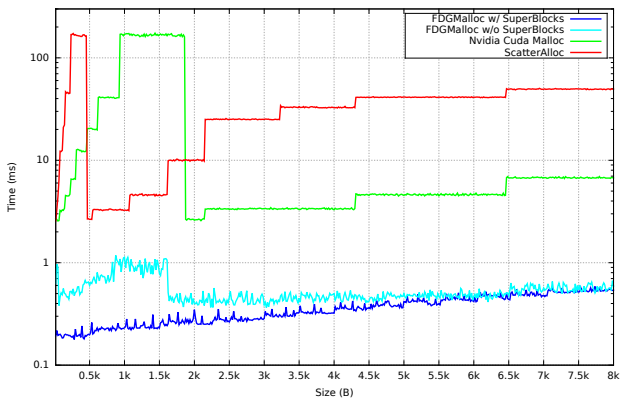

Figure 4: Performance Comparison of our proposed FDGMalloc, CUDAMalloc[11] and ScatterAlloc[12].

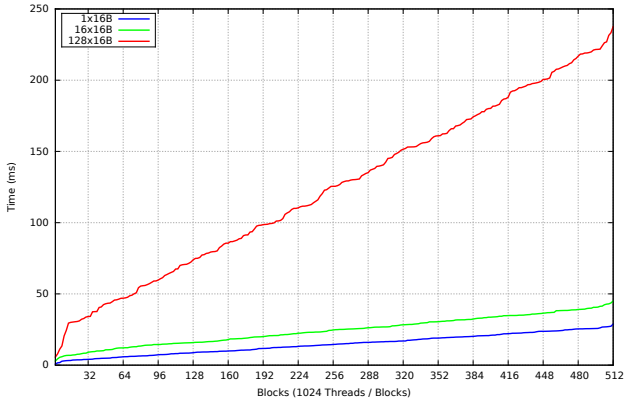

Figure 6: The chart shows a comparison of different allocation counts over the number of blocks using FDGMalloc with superblocks.

tion strategy and the usage of superblocks we implemented two different versions. One uses superblocks, the other one gathers all requests and allocates memory directly using the CUDA memory allocator.

Figure 4 shows the time needed to allocate S bytes. While CUDAMalloc and ScatterAlloc have to synchronize to serve the alloc request for each thread in a warp, we use the implicit synchronization between the threads. The chosen thread performs the request just by a simple atomic operation and propagates the pointer to the other participating threads. The graph shows that our allocator requires always the same amount of time to allocate a chunk of memory. Allocation time increases linearly because the operation needs more superblocks with growing memory consumption.

5.2 Varying thread count

In the second evaluation test case we vary the number of threads allocating a memory chunk. Because of the different amount of time needed to allocate different sizes of memory chunks the count and size of the allocations has been varied. In the end the mean of all results for one thread count has been calculated. The limited GPU Memory does not allow us to perform all tests with all combinations.

The values used for the tests have been X = { 1, 2, 4, . . . , 16384, 32768, 65536}, N = { 16, 32, 64, 128, 256, 512} and S = { 16, 32, 64, 128, 256, 512}. Figure 5 shows the results of this comparison as absolute (left figure) and relative (right figure, relative to FDGMalloc) values.

Since we allocate the first superblock during the initial phase our proposed FDGMalloc with superblocks is nearly ten times faster at the beginning than any other measured allocator. The speedup between both FDGMalloc (blue and light blue lines in Figure 5) is in the range of 10 to 300 depending on the number of threads requesting memory simultaneously. Here, one can clearly see the benefit of superblocks.

Comparing the default CUDA allocator (green line) with FDGMalloc without superblocks the gained speedup is related to the used voting function and reduction of concurrent memory requests. With a small number of threads (one to three) simultaneously requesting memory CUDAMalloc is faster. With more concurrent allocations the synchronization becomes a severe bottleneck.

5.3 Scalability

For the scalability evaluation of FDGMalloc with superblocks we used 1 to 512 blocks with 1024 threads each, which allocate 1, 16 and 128 times 16 bytes of memory ($X = 1024 * [1, 512]$, $N = \{1, 16, 128\}$, $S = 16$). Our results shown in Figure 6 clearly indicate the linear scaling of our allocator depending on the number of active threads. The bigger gradient in the range from 1 to 16 blocks is properly caused by hardware constrains. The GTX 680 has 8 multiprocessors which can handle up to 2 blocks with 1024 threads at once. With more blocks scheduled, the multiprocessors are able to hide the waiting time of a block by executing another block in the meantime.

6. CONCLUSION

We presented a dynamic memory allocator for many-core architectures with a wide SIMD width. Frequent and concurrent requests are reduced and handled efficiently by a voting function in combination with a fast allocation inside a superblock. The performance evaluations have shown that our proposed allocator is able to speed up dynamic memory allocation several orders of magnitude although it relies on the CUDA allocator. All in all we increase the SIMD scalability significantly for frequent dynamic memory allocations. The implementation shows that concurrent dynamic memory allocations in massively parallel architectures do not need to be slow. However, the assumption of the allocator do not allow it to be an all-round solution. It is still necessary to improve dynamic memory allocation schemes that allow memory to be arbitrarily freed during the execution. Our implementation is based on CUDA but could be extended to any of the other hardware architectures supporting a voting function. In the future we would like to evaluate the performance using Intel's Intel SPMD Program Compiler (ISPC) using AVX vector units and Xeon Phi.

7. ACKNOWLEDGMENTS

The work of S. Widmer and D. Wodniok is supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at Technische Universität Darmstadt.

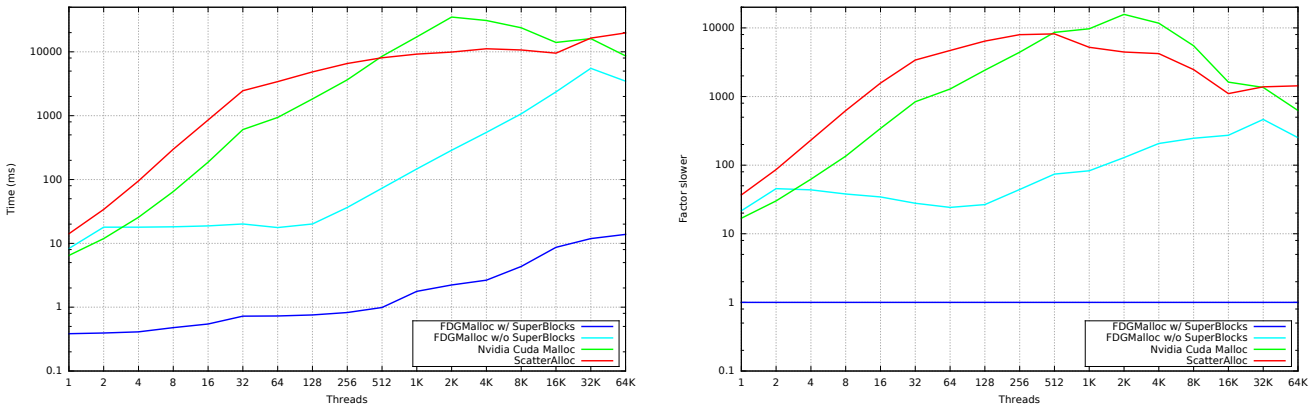

Figure 5: The figures show the performance comparison under varying thread count. On the left side with absolute values in milliseconds and the slowdown relative to FDGMalloc on the right.

8. REFERENCES

[1] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 2000.

[2] D. Dice and A. Garthwaite. Mostly lock-free malloc. In *Proc. ISMM*, 2002.

[3] W. Gloger. Dynamic memory allocator implementations in linux system libraries. `http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html`, 1998.

[4] D. Häggander and L. Lundberg. Optimizing dynamic memory management in a multithreaded application executing on a multiprocessor. In *Proc. ICPP*, 1998.

[5] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W.-m. Hwu. Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *Proc. IEEE ICCIT*, 2010.

[6] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. Mcrt-malloc: a scalable transactional memory allocator. In *Proc. ISMM*, 2006.

[7] INTEL. Intel SPMD Program Compiler. `http://ispc.github.com/`, 2012.

[8] D. E. Knuth. *The art of computer programming: fundamental algorithms*. 3rd edition, 1997.

[9] D. Lea. A Memory Allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`, 1996.

[10] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proc. SIGPLAN PLDI*, 2004.

[11] NVIDIA. CUDA Compute Unified Device Architecture. `http://www.nvidia.com/object/cuda_home_new.html`, 2012.

[12] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg. Scatteralloc: Massively parallel dynamic memory allocation for the GPU. In *Proc. InPar*, 2012.

[13] A. S. Tanenbaum. *Modern Operating Systems*. 3rd edition, 2007.

[14] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proc. IWMM*, 1995.