TECHNISCHE UNIVERSITÄT DRESDEN

CENTER FOR INFORMATION SERVICES
& HIGH PERFORMANCE COMPUTING
PROF. DR. WOLFGANG E. NAGEL

Großer Beleg

Enhancements of the massively parallel memory allocator ScatterAlloc and its adaption to the general interface mallocMC

Carlchristian Eckert
(Mat.-No.: 3385607)

Professor: Prof. Dr. Wolfgang E. Nagel
Tutor: Dr. Michael Bussmann (HZDR), Dr. Guido Juckeland (TU Dresden)

Dresden, October 29, 2014

Contents

Nomenclature

API	Application Programming Interface
CAS	Compare And Swap
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
GPGPU	General Purpose GPU
HPC	High Performance Computing
ID	IDentifier
MHz	MegaHertz
MIC	Many Integrated Cores
NoOp	No Operation
OOM	Out Of Memory
PIConGPU	Particle In Cell on GPU
RAM	Random Access Memory
SIMD	Single Instruction Multiple Data

1 Introduction

Dynamic memory allocation is one of the core features that is supported by most modern programming languages. It allows a program to request a portion of memory from the system at runtime and, thus, frees the programmer from the requirement to define the size of data structures at compile time. However, this flexibility comes at the cost of providing some form of memory management at runtime, which can cause performance penalties, increased fragmentation and other issues. Numerous different algorithms were developed to tackle these problems and resulted in well-established solutions for traditional single CPU systems [14, 3, 12].
One of the recent developments in computer hardware introduces additional processing elements, called accelerators, to augment the traditional concept of a single CPU with a co-processor that is capable of massively parallel data processing. Due to the novelty of these devices, memory allocation algorithms and their implementations are often not yet technically mature and, thus, suffer from non optimal performance characteristics [16, 17, 29, 33]. Therefore, innovative approaches to improve parallel memory allocation are a worthwhile endeavour.
The software PIConGPU [4, 28], a particle-in-cell code for fully relativistic particle simulations in plasma physics, exploits the parallelism of current high performance cluster systems, in particular by offloading most of the processing to the available accelerators in the form of GPUs. To cope with the massive amount of particles, storing data in the main memory of the accelerator is required. Currently, the performance problem of dynamic memory allocation is avoided through a custom heap buffer implementation, that relies on allocating large memory chunks of a fixed size.
The downside to this approach is the lack of flexibility. The maximum size of each data structure has to be estimated when allocating the buffer, so that it can fit all the data generated during the simulation. Since the simulation results are not always known in advance, this represents an undesirable limitation.
A possible solution is offered by *pool-based memory allocators*, which combine the idea of large pre-allocated memory areas with the capability of dynamically requesting portions of this pool. This work will show that, from all the considered implementations, ScatterAlloc [29] is the most promising one. However, the implementation provided by Steinberger et al. is aimed only at GPU-based accelerators from NVIDIA and requires a configuration through explicit C++ preprocessor macros.
This work aims to make the algorithm behind ScatterAlloc available for PIConGPU, so that it can be used to solve the mentioned challenges of the project. Therefore, the interface *mallocMC* is developed to provide a seamless integration of memory allocators into existing projects and ScatterAlloc is adapted to implement this interface in a modularized fashion. In case of future algorithms that are more suitable, the according part of the implementation can easily be replaced while keeping the interface stable. This allows for compatibility even in the face of changing hardware specifications. In addition, the reference implementation of ScatterAlloc is extended to allow for a more flexible configuration and an auxiliary feature to report remaining space on the accelerator.
The resulting memory allocator offers more features than NVIDIA's reference implementation, can replace the current buffer implementation of PIConGPU with no notable impact on performance (section 5.2) and can easily be included into other existing software projects.
The remainder of this work will be organized as follows: Chapter 2 gives an overview about accelerators, memory allocation and related work about allocators both on CPU and GPU. A generic interface to the chosen algorithm is provided in chapter 3, followed by chapter 4 which describes details of the actual implementation and the available features. Chapter 5 offers benchmark results to evaluate the performance of the implementation. Lastly, the achieved results of this work and possible improvements for the future are summarized in chapter 6.

2 Current state of the art

This chapter will introduce the reader to the concept of accelerator based programming, where additional processing units are available besides the traditional CPU and can be utilized as a computational resource. Later, it will provide an introduction to memory allocation in general, followed by a list of critical points that need to be considered when dealing with memory allocation on accelerators. The chapter closes with a discussion of existing memory allocators and will assess their suitability for massively parallel programs on accelerator hardware.

2.1 Accelerator based programming

According to Moore's Law, the density of transistors on a given die size doubles approximately every two years [10]. During the last decades, this resulted in continuously increasing single thread performance through a series of improvements in clock frequency, cache size, pipeline depth and others. While, despite many predictions, Moore's Law remains valid, the possible performance gains for a sequential instruction flow are stagnating [31]. In order to exploit the available transistors, processor design nowadays focuses on multiple identical processing units on the same chip.
In the wake of this paradigm shift, a new class of processing elements emerged that is completely dedicated to a massively parallel hardware design. These so-called *accelerators* are often used as coprocessors besides a traditional CPU. The first of these coprocessors evolved from graphics cards (GPUs) and even today, many accelerators are based on chips that are also used in high end GPUs that are even equipped with their own memory [9, p. 11f]. Since their use case is more general than that of a conventional GPU, these accelerators are often referred to as general purpose GPUs (GPGPUs).
Based on this legacy, the accelerator often does not execute the whole program itself, but is controlled by the CPU. The CPU and system memory represent the base system, called the *host*. The workflow for such an architecture, called *offloading*, is depicted in fig. 2.1: The host is responsible for sequential parts of the computation, since a modern CPU achieves higher performance for single threaded or moderately parallel workloads due to the higher clock frequency, pipeline depth and cache size relative to the accelerator. Once the program flow reaches a massively parallel section, the host copies the necessary data structures to the accelerator's memory and launches a function (*kernel*) on the accelerator itself. This kernel handles the according section in a highly concurrent manner and returns control back to the host, which can then retrieve the computed results from the accelerator.
Presently, the described workflow is the most commonly used way to interact with accelerators. However, some manufactures like Intel advertise accelerators that are based on an architecture of interconnected CPUs rather than graphic processors (Intel MIC [18]). These units present a completely different hardware approach and support not only the illustrated offloading mechanism, but are capable of executing entire programs without the support from the host.

2.2 Memory Allocation

In modern computer systems, usually more than a single process is active at any given time. This implies that the available hardware resources need to be shared efficiently between all active programs, in order to achieve a high utilization of the system. One of these resources is main memory, which can be reserved by programs through a process called memory allocation to store data structures at execution time.
Modern low level programming languages allow the programmer to handle this process of memory management explicitly through static and dynamic allocation [30, p. 277-283]. Static allocation determines

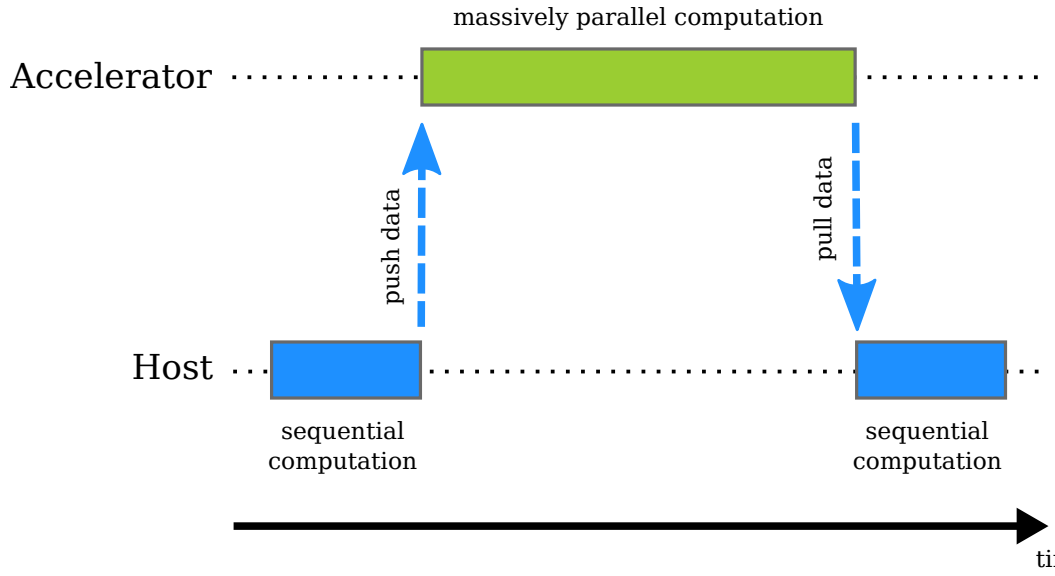


Figure 2.1: The host controls the general flow of the computation and can offload some suitable workloads to the accelerator hardware.

already at compile time, how much memory is needed and reserves the necessary amount when the program starts. Dynamic memory allocation on the other hand is performed at run time and uses memory from the so-called *heap* of the operating system [32, p. 173-185]. The software issues a system call to the operating system, which will use a *memory allocator* that searches for a continuous *chunk* of memory in the heap and serves the request by reserving the chunk and returning a pointer to that location. Once the program does no longer need this allocated *chunk* of memory, it is required to explicitly deallocate it in order to make the memory available again.
The heap itself is often a single resource that can be accessed by all processes[32, p. 754-758]. On the one hand, this structure leads to a lot of flexibility, since even a single process could allocate the entire memory if needed. On the other hand, the concept of a single resource introduces contention when dealing with parallelism, since the access to this resource is most often sequential. The available mechanisms to ensure this sequential access often suffer from contention when dealing with a high degree of concurrency, which leads to longer access times and results in lower performance of the program.

2.3 Critical points in memory allocation on many-core architectures

During the last decades, a multitude of memory allocators were written for various platforms (see section 2.4). The high number of implementations is also based on the fact that no implementation can fit all use-cases. This section will list typical requirements for memory allocators in today's many-core environments.
To understand the desired properties, it is important to look at the workloads that are currently handled with accelerators: Thousands of concurrent workers are active at the same time, each executing the same code on different data.[1] This code is often a time-critical part of a large program. Therefore, it should terminate as soon as possible. If each worker allocates memory, this results in thousands of allocations in a short time. Usually, these allocations are rather small, since each worker handles only a small workload and the amount of memory on accelerators is still rather limited. Moreover, the memory layout itself can have a considerable influence on speeds, due to effects like bandwidth utilization and cache efficiency.
A suitable allocator for accelerators should therefore be optimized towards **memory utilization** and

[1] based on Flynn's taxonomy [13], this pattern of a single instruction on multiple data is called *SIMD*.

speed for a workload consisting mostly of **small objects**. These goals are mainly influenced by the following criteria:

Scalability. Following the above-mentioned trend of parallelism in digital processors, concurrency in accelerators is expected to steadily increase in the future [11, 21]. The chosen allocators and algorithms need to be scalable even in the face of this development to keep up with the growing demands of embarrassingly parallel applications.

No false sharing of cache lines. In multi-core systems, multiple processing units often share a common cache. When workers access memory concurrently, a cache line will sometimes hold data from two or more different workers that are otherwise uncoupled and follow a different execution path, given the requests are small enough. This coupling of otherwise unrelated data can hinder performance and must be avoided. Therefore, it is required that each cache line is only owned by one single worker at a time or workers that can utilize the data together, to avoid this false sharing [20].

Low internal fragmentation. Internal fragmentation describes the ratio between allocated memory and used memory. This ratio increases as the allocator uses more memory to satisfy a request than is strictly necessary. This can arise due to hardware limitations (for example if alignment to full 16-byte addresses is necessary, a request of 8 byte will cause 16 byte to be reserved) or if the allocator can serve memory only from a pool of pre-allocated chunks, which are too spacious for the request. These empty spaces can waste precious memory on the accelerator. Furthermore, if fragments are smaller than the word size of memory accesses, this fragmentation can also negatively impact access speeds, since more bandwidth is consumed and the cache is unnecessarily filled.

Low external fragmentation. External fragmentation describes the average number of pieces that make up one memory request. If a request exceeds the size of the largest continuous area of free memory, it may be split into multiple smaller fractions which are distributed to the remaining free areas. The resulting fragmentation is undesirable, since it increases the number of memory accesses in case the fractions are smaller than the load size of the hardware architecture, thereby reducing the effective bandwidth. Furthermore, the resulting memory pattern looks increasingly fragmented, which potentially enables negative effects like false sharing.

Low contention between threads. Contention can arise when multiple workers try to allocate memory in parallel. Since the main memory is one single resource, the allocations have to be serialized in order to avoid conflicts. This sequential pattern conflicts with the concurrent nature of accelerators and should be minimized or avoided. Typical solutions on multi-core CPUs divide the memory into individual heaps for each worker [3]. However, this approach fails for the high number of workers present in accelerators, since the resulting heaps would be too small to be useful.

Support for SIMD execution. Most accelerators not only offer a high number of independent processing elements, but also SIMD-style instruction level parallelism. To fully exploit these features, it is necessary to reduce the aforementioned contention on certain hardware: If multiple workers execute code in a SIMD fashion, they will all compete for a resource at the same time, amplifying the problem even more [16]. As a possible worst case scenario, a group of n workers executing code as a SIMD group requests a resource n times, dismissing the benefits of SIMD execution entirely.

Hardware independence. The field of accelerators is still rather new and developing dynamically. A common hardware platform, as for traditional x86 based systems, does not yet exist. Therefore, it is desirable to adapt the allocator to new hardware without changing the whole program.

Flexibility in allocation size. As stated above, many algorithms demand allocation for small chunks of memory. The actual size of a chunk, however, can vary for at least two orders of magnitude,

leaving the need for a more precise configuration. Also, to avoid artificial restrictions upon the programmer, larger chunks should at least be possible.

Access speed. On traditional architectures, the time required to access a value located in main memory is orders of magnitudes longer than the time needed to take a value from an internal register of the CPU [15, p. B-3]. This constantly widening processor-memory performance gap is usually mitigated by the use of caches. On accelerator architectures, the vast amount of processing units and comparatively small caches per processing unit [9, p. 4, p. 332] emphasize the necessity of compute-heavy algorithms with few memory accesses. If such an access is unavoidable, it must be as fast as possible or it will become a bottleneck for the application.

No Blowup. If an application repeatedly reallocates memory, some allocators struggle to re-use previously freed chunks of memory due to their internal heap structures and parallel access patterns. If this happens, the overhead of unusable memory slowly increases over time. Especially in the face of high fragmentation and concurrency, this can be a problem on accelerators and needs to be avoided.

2.4 Related Work

As pointed out in section 2.2, the management of dynamically allocated memory is performed by a memory allocator. Allocators can now be classified based on the way their heap is organized: Arguably the most basic kind is created with single-threaded applications in mind and uses a single heap that is accessed by the only existing thread. The classic implementation of GNU Malloc from Doug Lea [22] would be an example. Benefits include the relative simplicity of the implementation, resulting in a lower probability of programming errors, and the low fragmentation that results from serial access. However, due to the single-threaded nature inherent to this serial single-heap design, this approach does not allow for any scalability in the face of multiple threads.
A straightforward way to deal with the problem of serial single-heap allocation is to increase the number of heaps to match the number of processing units. Classic problems with this approach include false sharing of cache lines [3] as well as only limited scalability. The latter is caused by the fact that memory is a limited resource: If a high number of processing units each manages a private section of this memory, the sections tend to become smaller as the number of processing units increases. In case of massively parallel hardware, the resulting heaps would be way too small to be useful.
The current generation of memory allocators for accelerators therefore often employs only a single heap instead, but allows concurrent access of multiple workers to the same heap which avoids the mentioned problems. However, access to the heap can be costly, since it is often regulated by some mechanism like locking or atomic compare and swap (CAS). To reduce fragmentation and improve performance, these class of algorithms is often tuned to a specific use-case and offers only a limited number of different allocation sizes.
The rest of this chapter will introduce several popular memory allocators to provide an overview about existing research. According to the criteria in section 2.3, the suitability of these allocators for accelerators is evaluated.

2.4.1 Ptmalloc

Ptmalloc [14] was developed by Wolfram Gloger and is the current allocator of the Linux GNU C library. It extends Doug Lea's serial single heap allocator to a multi heap allocator, where the heaps are connected as a single linked list. Benchmarks conducted by Berger et al.[3] find that its allocation speed scales well up to six hardware threads. It is still considered a viable implementation on current single-CPU systems, being hardware independent for all major CPU architectures and offering flexible allocation sizes with low fragmentation. However, in certain applications it is easily outperformed by more optimized allocators like Hoard (section 2.4.2) and the support for concurrency is not sufficient for accelerators.

2.4.2 Hoard

Hoard [3] by Berger et al. was specifically designed with multi-processor systems in mind. It was one of the first implementations to address the problems of unbounded blowup and false sharing. Each thread is assigned to a private heap, which holds multiple superblocks to organize the data. Each of those superblocks can only hold memory chunks of a fixed size. This approach improves caching, access time and fragmentation. To balance the memory needed by each heap, Hoard also provides a global heap. If the private heap of a thread runs out of memory, it can request a new superblock from the global heap or return superblocks after memory becomes free. The allocation strategy is optimized for small memory accesses, but retains the flexibility to allocate large chunks of memory by directly utilizing the platform's memory subsystem. According to Berger et al. [3], Hoard manages to avoid fragmentation and false sharing while simultaneously maintaining strong scaling to up to 14 hardware threads. For workloads typical in scientific simulations, it outperforms all CPU-focused solutions examined in this work. However, the used algorithm requires one lock when allocating memory and usually two locks when freeing the memory again. Global, lock-based mechanisms pose a natural limit to scalability on massively parallel systems. However, locking becomes even more problematic in the face of SIMD-style execution and a technique called *simultaneous multithreading* (SMT), which are both common principles in modern accelerator cards. The small possible heaps when using thousands of threads concurrently emphasize the scalability problems even more.

2.4.3 CUDA toolkit allocator on device

On CUDA capable GPUs, NVIDIA offers a reference implementation for memory allocation. This memory allocation is available for GPUs with a compute capability of at least 2.0 [24] and can be used by calling `malloc()` directly on the accelerator. As an implementation detail, calling `new` or `new[]` will also invoke the system call to `malloc()`. According to several benchmarks, this allocator seems to be optimized towards fewer allocations of rather large chunks of memory. It does not require special configurations, achieves a rather low fragmentation and scales better than regular CPU based algorithms would. However, scalability struggles to keep up with an increasing number of active workers [16, 29, 33]. Especially in embarrassingly parallel scenarios, memory allocation with the provided implementation presents a bottleneck. This behaviour has inspired multiple research groups to implement their own memory allocator that is better optimized for high-performance workloads.

2.4.4 PIConGPU HeapBuffer implementation

For the PIConGPU project, the apparent problems with the CUDA toolkit allocator resulted in the development of a custom memory management system that works like a ring buffer. For each type of particles (*species*), a fixed amount of memory is allocated from the heap when the simulation starts. This large pool is then divided into a number of smaller *frames* of a fixed size to form the *GridBuffer*. For each GridBuffer, a *ring buffer* is created that can hold the indices of the frames. Figure 2.2 illustrates the memory layout for a single species: The *push* pointer indicates the position of the first chunk that is currently used. The *pop* pointer points to the spot in the ring, which contains the ID of the first available free chunk in the GridBuffer. When a worker requests a new empty frame, it gets the position of the next frame from the pop pointer and said pointer is advanced one position in the ring buffer. Likewise, when deallocating memory, a worker simply needs to write the index of the freed chunk to the position of the push pointer and advance the pointer afterwards.
The obvious benefit of this allocator lies in its simplicity. Each movement of pop or push requires only a single atomic operation, followed by a lookup in the global memory of the GPU and some boundary checks to recognize an underflow of the ring buffer. This leads to simple, easily maintained program code as well as high performance, since atomic operations prove to be fast enough not to impact program execution time.

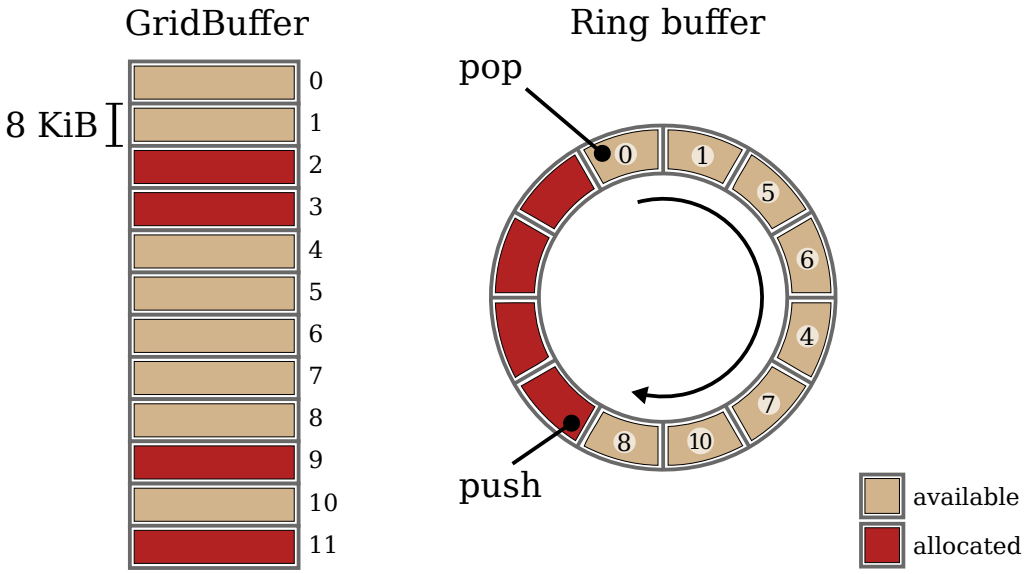


Figure 2.2: The ring buffer is used to store the IDs of frames in the GridBuffer. The pointers *pop* and *push* are used to get the ID of the next free frame to process and to write the IDs of newly deallocated frames.

The most severe downside comes with the inflexibility of allocating the GridBuffers in fixed sizes: If more than one species is used, the memory of the GPU has to be divided among the species according to the ratio of particles for each species. If the ratio changes during the simulation, the fixed pools can not account for this shift and memory will be wasted. Furthermore, if the memory of a GridBuffer becomes completely filled, this will be visible only to the very next worker trying to pop an empty element from the ring buffer due to implementation details. This leads to the second downside, namely the possibility that workers miss the fact that no more memory is available. As a result, workers might corrupt the memory of other workers.

This HeapBuffer implementation is currently used in PIConGPU. However, the mentioned downside of fixed allocation sizes limits the simulations to a maximum of two species. Therefore, the project requires a new allocator that is able to manage the allocation of frames in a flexible way.

2.4.5 XMalloc

To address the exisiting performance limitations of the default CUDA toolkit allocator (section 2.4.3), Huang et al. developed XMalloc [16, 17], a memory allocator specifically optimized for accelerators. It utilizes the idea of superblocks as they were introduced by Hoard to reduce fragmentation, by optimizing towards certain chunk sizes which are sorted into the superblocks. Unlike Hoard, which uses a single global heap, these superblocks are managed in a hierarchy of buffers to improve access speeds by caching. Threads can access superblocks directly, without managing a private heap of their own. Instead of locking, XMalloc uses special *compare and swap* (CAS) instructions of the hardware, to improve performance and reduce contention. It also addresses the problem with SIMD architectures, where all workers inside a SIMD-unit compete for the same resource: All members of the SIMD-group announce the desired allocation size for a request. These sizes are then combined and only one worker in the SIMD group executes the allocation. Therefore, only this worker needs to issue the CAS instruction. The implementation provided by Huang et al. is designed specifically for NVIDIA CUDA graphics cards and, according to the authors, it achieves a significant speedup for many workloads, as compared to plain usage of the built-in CUDA allocator.

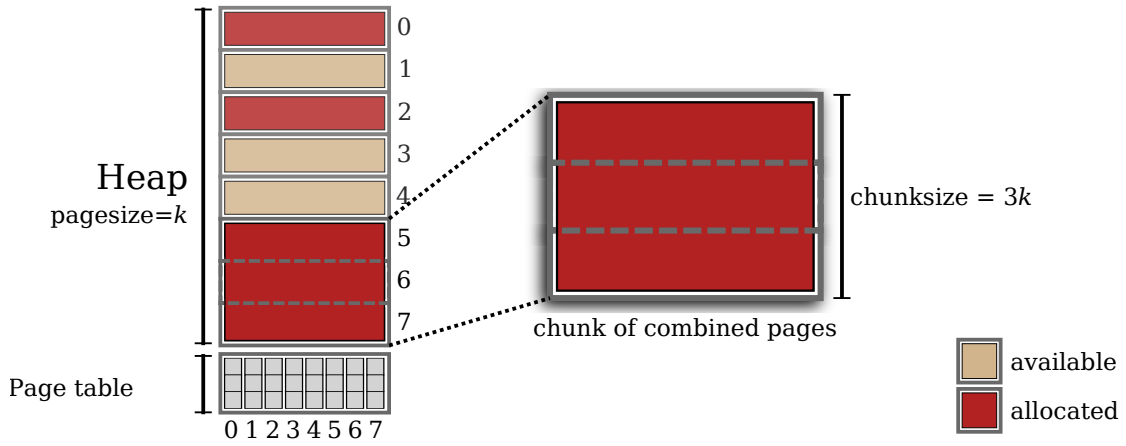


Figure 2.3: If the chunk size exceeds the size of a single page, multiple pages can be combined and allocated as a single chunk. To reduce fragmentation, these large chunks are located at the end of the heap.

2.4.6 ScatterAlloc

In order to improve the success that came from XMalloc, Steinberger et al. created ScatterAlloc [29]. Again, this memory allocator is tailored specifically to hardware from NVIDIA. The algorithm combines the idea of a single global heap as seen in Hoard with the SIMD optimizations from XMalloc. Unlike XMalloc, it avoids a single point of serialization in order to improve concurrency with very high numbers of active workers. Instead, the accessed positions in the heap are determined by a hash function that scatters the allocations over the available heap to avoid concurrent access in the same memory pages.
The fill levels of the heap and each single page are stored by a page table that holds internal chunk size, the number of allocated elements, and a bitfield to track the positions of allocated chunks of that page.
Instead of relying on fixed object sizes, the chunk size for each page is set by the first worker that accesses it. Depending on the chunk size, the page is split in different ways to reduce internal fragmentation for each page:
For very large requests that do not fit into a single page, multiple pages can be concatenated. Figure 2.3 depicts a case where three pages are combined to form a single chunk. Since a consecutive number of pages is required, only a single worker is able to search the heap for suitable positions. Therefore, this allocation scheme is unsuitable for highly parallel workloads.

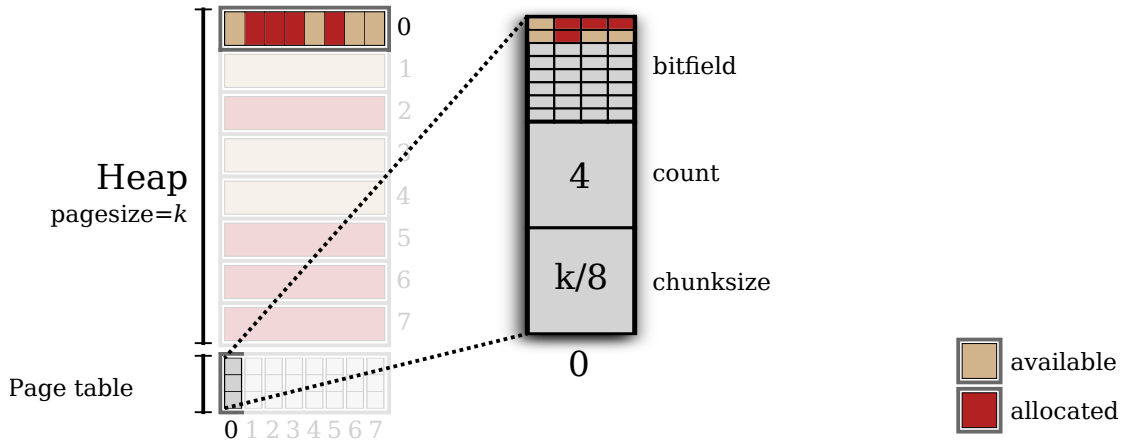


Figure 2.4: The chunk size on page 0 equals $\frac{1}{8}$ of the page size, which leads to a page layout without hierarchical bitfields. The page is split into eight equal slots of which four are currently used. The corresponding bitfield in the page table keeps track of the allocated positions.

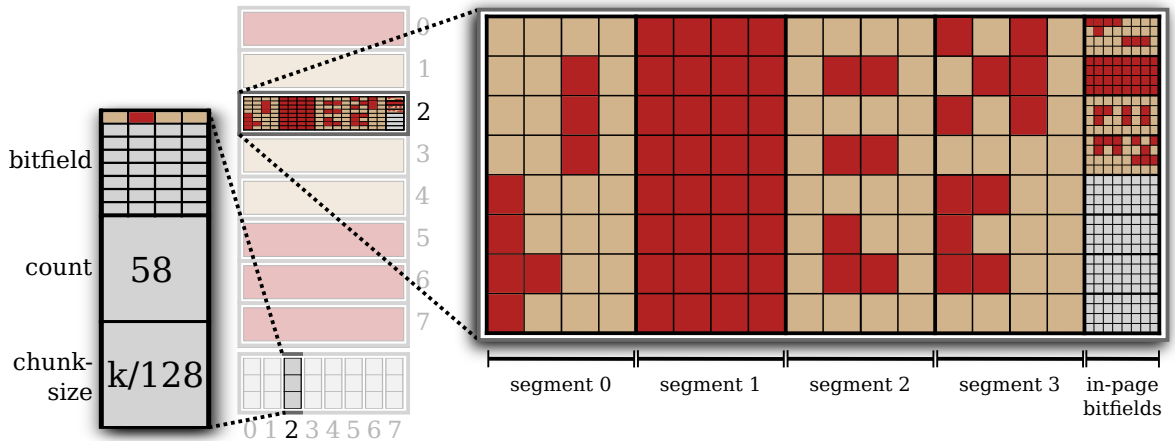


Figure 2.5: The chunk size on page 2 equals $\frac{1}{128}$ of the page size, which leads to a page layout with hierarchical bitfields inside the page itself. These bitfields are located at the end of the page and can each track 32 chunks. The bitfield in the page table now indicates, when an in-page bitfield is completely filled.

If the chunk size allows for a maximum of 32 chunks in a page, the chunks are simply stored sequentially inside the page. Figure 2.4 illustrates such a layout for a given page size k and a chunk size $s = \frac{k}{8}$ on page 0. The bitfield in the page table corresponds directly to the positions of allocated chunks. Since only eight chunks fit into a single page, the last 24 bit of the bitmask are not used.
For smaller chunk sizes, a hierarchical structure is generated that enables a page to store up to 1024 objects. The hierarchy is set up directly inside the page through up to 32 bitfields, each capable of tracking the status of 32 chunks. The bitfield in the page table itself is then used to indicate if one of the internal bitfields is completely filled. In fig. 2.5, 128 chunks can fit into a single page. Since the 32 bit of the page table are insufficient to manage that many chunks, bitfields are created directly at the end of the page. These in-page bitfields now track the allocation for their corresponding segment of chunks. The bitmask in the page table is used as a record for the four segments, indicating that the second one is completely filled.
Although ScatterAlloc can introduce a comparatively high fragmentation, the trade-off can be beneficial when the large benefits for scalability, contention and access speed are taken into account: in combination with the reduced time to search for free memory due to the scattering formula, the authors of ScatterAlloc find it to be about 10 times faster than XMalloc and 100 times faster than the default allocator provided by the CUDA toolkit.
As a result of the unprecedented allocation speed of ScatterAlloc, this algorithm was chosen to be a suitable replacement for the HeapBuffer (section 2.4.4) in PIConGPU. Furthermore, it will be adapted to a new interface, *mallocMC*, as hinted in chapter 1.

3 Interfaces

This chapter introduces the reader to the concept of policy based design and presents an interface to the memory allocator mallocMC based on this idea.

3.1 Policy Based Design

The existence of fundamentally different accelerator concepts (see Section 2.1) emphasizes the benefits of a modular design: The hardware dependency can be implemented inside a module. If the hardware architecture changes, the program can simply be adapted by offering a new implementation of said module to substitute the affected one, satisfying the requirement **hardware independence** from section 2.3.
Furthermore, the proposed memory allocator mallocMC aims to offer an extensible platform to generate a custom memory allocator which can be adapted not only to varying platforms, but also to different use-cases through flexible configuration of its components. This creates the possibility to tune performance and the used algorithms towards the hardware and workload, thus, allowing for portability in the future.
A popular way to implement the aforementioned features in the C++ programming language was mentioned in [2, p. 3-21] as *policy based design*. This is basically a variant of the strategy design pattern [1, 6, 26] that takes effect at compile time rather than runtime, thus, resulting in higher performance. In particular, policy based design suggests to split different aspects of the functionality into so called *policies*, each defining a fixed interface and a responsibility. These responsibilities should ideally be mutually *orthogonal*. This property is useful to enable polymorphism over multiple implementations of a policy. The actual implementation of a policy is called a *policy class* and provides a certain functionality that is accessible through the prescribed interface. The different policies are then to be combined through a so called *host class*[1] that supplies the necessary glue logic to merge the responsibilities into a meaningful context. Therefore, the instance of a host class displays a behaviour that depends on the actual policy classes inside. In the case of orthogonal policies, the behaviour can be influenced by simply replacing one policy class by another implementation of the same policy.
Figure 3.1 depicts such a policy based design, where the host class inherits from two policies, *PolicyA* and *PolicyB*. Each of these policies is implemented by two actual policy classes that fulfil the interface of their respective policy. Depending on the use case, the programmer is able to decide which combination of behaviours of PolicyA and PolicyB is required.

3.2 HostClass: mallocMC::Allocator

The allocator **mallocMC::Allocator** is implemented as a class template of the five types **CreationPolicy**, **DistributionPolicy**, **OOMPolicy**, **ReservePoolPolicy** and **AlignmentPolicy**.
Each of these policies is geared towards a specific purpose: ReservePoolPolicy handles a pool of raw memory on the accelerator, whereas the internals of that pool are managed by CreationPolicy. The AlignmentPolicy governs alignment and padding of raw memory as well as internal structures. The DistributionPolicy can modify memory requests before they are actually allocated by CreationPolicy as an internal structure. OOMPolicy[2] will become active if the CreationPolicy failed to allocate space in the internal structures.

[1] In the context of contemporary high performance computing (HPC), the term host class can be slightly misleading, since it is used in several different contexts. In the following, *host class* always refers to a structure implementing logic around a policy based design. *Host*, however, refers to the hardware entity complementary to the *accelerator*.
[2] OOM here stands for **O**ut **O**f **M**emory.

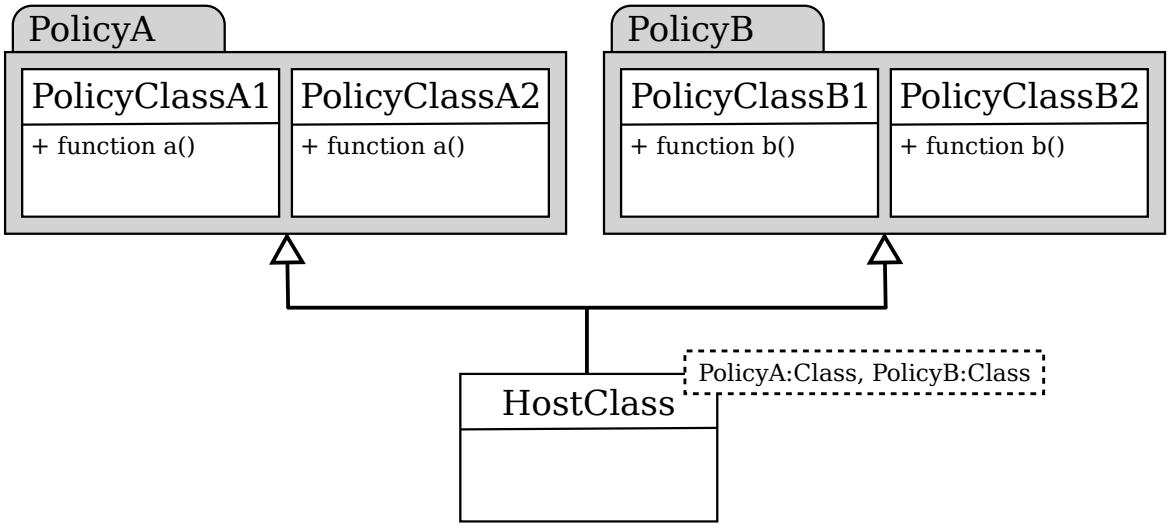


Figure 3.1: A policy based design. The host class inherits from two policies, *PolicyA* and *PolicyB*, that provide disjunct functionalities. The actual policy class for each policy is supplied to the host class as a template argument at compile time.

The allocator utilizes the interfaces of the different policies to compose its own external interface. This interface exposes the complementary functions **`alloc()`**/**`dealloc()`** and **`initHeap()`**/**`finalizeHeap()`**. Apart from that, it will provide the optional functions **`getAvailableSlots()`** and **`info()`**, if their respective preconditions are fulfilled.

```
void* alloc(size_t bytes)
```

Will be executed by code running on the accelerator. Takes a number of bytes to allocate. Should allocate at least **`bytes`** byte of memory, and return a pointer to it. The pointer to the memory area is aligned with **`AlignmentPolicy::alignAccess()`** and checked for success.

```
void dealloc(void* p)
```

Will be executed by code running on the accelerator. This function is complementary to **`alloc()`**. It takes a pointer to a memory area that was allocated by **`alloc()`** and may free the memory, possibly making it available for another allocation by **`alloc()`**.

```
void* initHeap(size_t bytes)
```

Will be executed by code running on the host. The input represents the size of the heap in bytes. Will allocate and create all the necessary structures to use **`alloc()`** and **`dealloc()`** on the accelerator later on. Naturally, this operation will require at least **`bytes`** bytes of unused memory on the accelerator. Returns a pointer to the allocator object on the accelerator, if possible.

```
void finalizeHeap()
```

Will be executed by code running on the host. This function is complementary to **`initHeap()`** and can undo the effects of **`initHeap()`** to reset the memory state on the device. Naturally, this should increase the amount of unused memory on the device by the size of the heap. As a side-effect, all pointers to memory allocated on the accelerator through **`alloc()`** will be invalidated.

```
unsigned getAvailableSlots(size_t slotSize)
```

The result of this function is only valid, if the chosen CreationPolicy supports it, as indicated by `CreationPolicy::providesAvailableSlots`. There may be different implementations to allow access from the host as well as from the accelerator. Given a fixed size `slotSize` and a reference to the instance of the allocator, it should return the total number elements of size `slotSize` that can still be allocated on the heap. Multiple successive calls to this function must return the same result if the heap did not change.

```
std::string info(std::string linebreak = "")
```

This function only exists, if all supplied policies implement the optional `classname()` member function. It will be executed by code running on the host. This function will print information about the used policies, utilizing the `classname()` method of each policy. It takes a parameter `linebreak` which will be used to separate the information from the different policies. Since this is intended for humans, the output format has no fixed constraints or definitions and is subject to change. This function does not change or use any internal state. Calls to this function on the same allocator instance and same input will always yield the same result.

3.3 ReservePoolPolicy

ReservePoolPolicy prescribes a class that exposes two complementary member functions `setMemPool()` and `resetMemPool()`. The implementation may also define an additional member function `classname()`. The class aims to acquire the memory area (the `pool`) that will be used to contain the heap. The chosen CreationPolicy is responsible to create the necessary data structures inside this memory area.

```
static void* setMemPool(size_t memsize)
```

Will be executed by code running on the host. Needs at least `memsize` bytes of memory available on the accelerator. Reserves/allocates `memsize` bytes as the pool on the accelerator. After this function succeeds, there will be `memsize` bytes of memory occupied by the pool. Returns a pointer to this pool of memory, if applicable. Implementations like `cudaDeviceSetLimit()` from the CUDA runtime API do not offer an addressable pool object.

```
static void resetMemPool(void* p)
```

Will be executed by code running on the host. Can take a pointer to a memory pool acquired through `setMemPool()`. After this function returns, the effects of `setMemPool()` will be reset and memory on the accelerator used by the pool will be available for reallocation.

```
static std::string classname()
```

Optional function. Will be executed by code running on the host. Should return a human readable identifier for the policy as a string, perferably without any newlines. The string might contain the name of the policy and used template arguments. This is meant as a representation for humans. Therefore, no particular formatting is required. This function must not change any state and must always return the same result.

3.4 AlignmentPolicy

AlignmentPolicy prescribes a class that exposes the static member functions `alignPool()` and `applyPadding()`. The implementation may also define an additional member function `classname()`.

This policy aims to ensure that memory accesses are aligned to exploit more efficient behaviour of the hardware. This can avoid false sharing (see section 2.3), if the size of a memory access is always a multiple of the cache line size. This policy must not change the state of the heap once it is initialized.

```
static boost::tuple<void*, size_t> alignPool(void* memory, size_t memsize)
```

Will be executed by code running on the host. Takes a pointer to a chunk of memory (the pool) and the size of this chunk. The state of the pool is uninitialized, so there are no internal heap structures set up which might get corrupted. Returns a possibly modified pointer to the same pool along with a possibly modified size of the pool.

```
static uint32_t applyPadding(uint32_t bytes)
```

Will be executed by code running on the accelerator. The input `bytes` represents the size of a memory request in bytes that should be padded to a possibly larger size. Returns the size of a request in bytes after padding has been applied. This number may differ from the input as needed.

```
static std::string classname()
```

Optional function. Will be executed by code running on the host. Should return a human readable identifier for the policy as a string, perferably without any newlines. The string might contain the name of the policy and used template arguments. This is meant as a representation for humans. Therefore, no particular formatting is required. This function must not change any state and must always return the same result.

3.5 DistributionPolicy

DistributionPolicy prescribes a class that exposes two complementary member functions `collect()` and `distribute()`. The implementation may also define an additional function `classname()`. The class aims to provide a way of resizing a group of memory requests. This can be useful to exploit beneficial behaviour (e.g. only one worker in a workgroup allocates memory, since it is faster than many small requests). The SIMD optimizations discussed in section 2.3 can be expressed through this policy. Since the behaviour has to be reversible, it can be instantiated as an object to store internal state information about the resizing operation. This policy must not change the state of the heap.

```
uint32_t collect(uint32_t bytes)
```

Will be executed by code running on the accelerator. Takes the size of a memory request. The output is a potentially different size for this memory request, according to the applied optimization. Information about the used transformation can be stored as an internal state that is used later by `distribute()`.

```
void* distribute(void* memBlock)
```

Will be executed by code running on the accelerator. Takes a pointer to a memory area `memBlock` and uses the inverse of the transformation applied in `collect()` to modify the pointer. The output should be a potentially modified pointer into the input memory area.

```
static std::string classname()
```

Optional function. Will be executed by code running on the host. Should return a human readable identifier for the policy as a string, perferably without any newlines. The string might contain the name of the policy and used template arguments. This is meant as a representation for humans. Therefore, no

particular formatting is required. This function must not change any state and must always return the same result.

3.6 CreationPolicy

CreationPolicy prescribes a class that exposes five member functions `create()`, `destroy()`, `isOOM()`, `initHeap()`, and `finalizeHeap()`. The implementation may also define the three additional functions `classname()`, `getAvailableSlotsHost()`, and `getAvailableSlotsAccelerator()`. Lastly, it must provide a typedef `providesAvailableSlots`. The class aims to set up data structures in an existing memory pool to create a heap. This heap is used to satisfy memory requests of a given size (the size for each worker is determined by `DistributionPolicy::collect()`). The class uses its knowledge about the allocation strategy to check if the memory request of a worker was successful. The chosen algorithm can have a high impact on most requirements from section 2.3, especially **fragmentation**, **contention**, the **flexibility to allocate small objects** and **scalablity** in general.

```
static void* initHeap(mallocMCType& obj, void* pool, size_t memsize)
```

Will be executed by code running on the host. Establishes and initializes the necessary data structures to manage the heap. `obj` is a reference to the instance of the host class `mallocMC::Allocator`; `pool` points to an empty but allocated memory area of size **memsize** that will be used as a heap. It may be `NULL`, if no explicit pointer is needed for this policy. After the function returns, `pool` holds heap data structures and must not be modified directly by other classes. If the heap is directly addressable, it may be returned. This function is complementary to `finalizeHeap()`.

```
static void* create(uint32_t bytes)
```

Will be executed by code running on the accelerator. It takes a number of bytes that should be allocated on the heap. If this function succeeds, it will have allocated the requested memory on the accelerator. Should return a pointer to a memory region where at least `bytes` byte of memory are available. This function is complementary to `destroy()`.

```
static void destroy(void* mem)
```

Will be executed by code running on the accelerator. Takes the pointer to a memory area that was allocated by `create()` and can make it available for a new allocation. The pointer will be invalid after this function returns. Calling `destroy()` twice on the same pointer is undefined. This function is complementary to `create()`.

```
static void finalizeHeap(const mallocMCType& obj, void* pool)
```

Will be executed by code running on the host. `obj` is a reference to the instance of the host class `mallocMC::Allocator`; `pool` points to a memory area that keeps its heap. The structures are initialized through `initHeap()`. Therefore, this function is executed on an initialized pool and should destroy the data structures inside this pool, thereby undoing what `initHeap()` created. The allocation of the pool itself must not be affected by this. This function is complementary to `initHeap()`.

```
static bool isOOM(void* p, size_t req_size)
```

Will be executed by code running on the accelerator. Takes the pointer to a memory area that was allocated by `create()`. Must return true, if `p` appears to be the result of a failed memory request to `create()` with the size of `req_size`. This function must not change any state.

```
unsigned getAvailableSlotsAccelerator(size_t slotSize)
```

Optional function. The presence of this function in the implementation is indicated by the typedef `providesAvailableSlots`. Will be executed by code running on the accelerator. Given a fixed size `slotSize`, it should return the total number elements of size `slotSize` that can still be allocated on the heap. Multiple successive calls to this function must return the same result, if the heap did not change.

```
unsigned getAvailableSlotsHost(size_t slotSize, mallocMCType& obj)
```

Optional function. The presence of this function in the implementation is indicated by the typedef `providesAvailableSlots`. Will be executed by code running on the host. Given a fixed size `slotSize` and a reference to the instance of the allocator, it should return the total number elements of size `slotSize` that can still be allocated on the heap. Multiple successive calls to this function must return the same result if the heap did not change.

```
typedef boost::mpl::bool_<T> providesAvailableSlots
```

Typedef that indicates the presence of the functions `getAvailableSlots()`. The typedef can have different values for host and accelerator.

```
static std::string classname()
```

Optional function. Will be executed by code running on the host. Should return a human readable identifier for the policy as a string, perferably without any newlines. The string might contain the name of the policy and used template arguments. This is meant as a representation for humans. Therefore, no particular formatting is required. This function must not change any state and must always return the same result.

3.7 OOMPolicy

OOMPolicy prescribes a class that exposes a member function `handleOOM()`. The implementation may also define an additional member function `classname`. The class aims to handle failed memory requests.[3]

```
static void* handleOOM(void* memBlock)
```

Will be executed by code running on the accelerator. Will be invoked if the pointer `memBlock` appears to be the result of a failed memory request as defined by `CreationPolicy::isOOM()`. Therefore, this function is invoked in an exceptional situation where the system is no longer able to satisfy some memory requests. `handleOOM()` may throw exceptions and may change the given pointer. The function can be used to trigger behaviour to solve the memory shortage. The state of the heap after executing this function is implementation-defined.

```
static std::string classname()
```

Optional function. Will be executed by code running on the host. Should return a human readable identifier for the policy as a string, perferably without any newlines. The string might contain the name of the policy and used template arguments. This is meant as a representation for humans. Therefore, no particular formatting is required. This function must not change any state and must always return the same result.

[3] The name of this policy originates from an abbreviation of the term **O**ut **O**f **M**emory.

4 Implementation

This chapter is split into three sections. The first gives details about improving usability of mallocMC for the end user — primarily PIConGPU. The second section provides information about the implementation of the host class and how it combines the interfaces presented in chapter 3. As a reference implementation, the third section splits the high performance algorithm ScatterAlloc into policy classes to enable it as a possible configuration of mallocMC. Aside from just porting ScatterAlloc, it is enhanced through functions to check the amount of used memory as well as a more explicit way to provide configuration parameters and bug-fixes. These features amount to the intended implementation goal, allowing mallocMC to be used as a drop in replacement for current memory allocators while offering a lot of extensibility to support upcoming ideas.

4.1 mallocMC as drop-in replacement for existing allocators

Since the implementation is written in C++, the adaption to existing projects utilizes *namespaces* to encapsulate the allocator itself. Consequently, all classes reside in the namespace **mallocMC** or in a nested sub-namespace.
All the allocation functions are member functions of the class **mallocMC::Allocator** and require an instantiated object. However, usual calls to **new** or **malloc()** do not require an explicit object. Therefore, passing an additional object to most functions is required and complicates usability in comparison to the built-in solution. To minimize boilerplate code for the end user, a collection of macro definitions is made available. These definitions initialize a global object of the composed type (see section 4.3.7 for the configuration), which is then used for all subsequent calls to an allocator. Specifically, calling

```
MALLOCMC_SET_ALLOCATOR_TYPE(T)
```

instantiates a global object of type **T** and provides the following functions to access it:

```
void* mallocMC::initHeap();
void  mallocMC::finalizeHeap();

void* mallocMC::malloc();
void  mallocMC::free();

bool     mallocMC::providesAvailableSlots();
unsigned mallocMC::getAvailableSlots(size_t slotSize);
```

Unfortunately, the C++ programming language does not allow the operators **new()** and **delete()** to be defined inside a namespace. As a consequence, their definition is omitted here. Also, simply placing **new()** in the global namespace would override the already existing definitions. If, however, overriding of global implementations is actually desired, the macros

```
MALLOCMC_OVERWRITE_NEW()
MALLOCMC_OVERWRITE_MALLOC()
```

can be called to replace **new**, **new[]**, **delete**, **delete[]** and **malloc()**, **free()** respectively.[1]

[1] Current versions of CUDA treat **new()**, **new[]()** and **malloc()** as the same system call. It is therefore not possible to use the first of the two macros in such an environment. However, this implies that overriding **malloc()** will actually also override **new()** on this accelerator platform.

4.2 A unified host class for different allocators

The host class itself serves multiple purposes. Most importantly, it combines the five policies to form a useful allocator. The used policy classes must be given as template parameters when defining the type of the allocator (see section 4.3.7) and are either directly instantiated or inherited from.
Apart from that, there exists a traits structure to offer information about functions that are optionally implemented inside a policy. To keep the interface stable, the host class uses compile time polymorphism over functions as demonstrated in [2, p. 30f.]. Therefore, the implementation defines two functions that are overloaded with respect to one of their arguments. One of those functions implements the behaviour which is optional in a policy, whereas the other one provides a default behaviour in case the policy class does not implement the optional function. The *type* of the overloaded argument is used to choose at compile time which function will be used. The function **`getAvailableSlots()`**, which is optional for a CreationPolicy, is a good example:[2]

```
public:
unsigned getAvailableSlots(size_t slotSize)
{
  return getAvailSlotsPoly(slotSize,
      boost::mpl::bool_<CreationPolicy::providesAvailableSlots::value>());
}

private:
unsigned getAvailSlotsPoly(size_t slotSize, boost::mpl::bool_<false>)
{
  assert(false);
  return 0;
}

unsigned getAvailSlotsPoly(size_t slotSize, boost::mpl::bool_<true>)
{
  CreationPolicy::getAvailableSlots(slotSize);
}
```

The type **`providesAvailableSlots`** is defined in each implementation of a CreationPolicy and used to access the internal implementation only if it exists.
Lastly, the host class employs the external class **`PolicyConstraints`** to ensure already at compile time that defined constraints between non-orthogonal policy classes are met. It takes all supplied policy classes as template parameters and uses them to match internal trait-style structs:

```
// The default PolicyCheck2 (does always succeed)
template<class Policy1, class Policy2>
struct PolicyCheck2{};

template <class CP, class DP, class OP, class GP, class AP>
class PolicyConstraints
{
  PolicyCheck2<CP, DP> c;
};

template<class x, class y, class z>
struct PolicyCheck2<typename CreationPolicies::Scatter<x,y>, typename
    DistributionPolicies::XMallocSIMD<z> >
{
  BOOST_MPL_ASSERT_MSG(x::pagesize::value == z::pagesize::value,
  Pagesize_must_be_the_same_when_combining_Scatter_and_XMallocSIMD, () );
};
```

[2] For information how to use such an overloaded function safely, see A.5.

In the example, a `PolicyCheck2` is instantiated with a CreationPolicy `CP` and a DistributionPolicy `DP`. Should they refer to the policy classes `Scatter` and `XMallocSIMD`, the matching template specialization asserts that their configuration parameter `pagesize` is identical.
A programmer implementing a policy can add custom `PolicyCheck` specializations to handle possible restrictions for policy classes.

4.3 ScatterAlloc as reference implementation

To implement ScatterAlloc in terms of policies as needed by the host class, the algorithm has to be split to fit the interface defined in chapter 3. In this case, it is not possible to keep all policies completely orthogonal, as some constraints apply. Most importantly, this affects the configurations of the AlignmentPolicy `Shrink` and the CreationPolicy `Scatter`, since both implementations need to match in their parameter `pagesize`. This limitation is enforced by the constraint mechanism (see 4.2). Furthermore, ScatterAlloc was implemented for CUDA capable GPUs. Therefore, some of the policies presented in this section are currently restricted to NVIDIA hardware. Although a more generic implementation might seem desirable under the aspect of **hardware independence** (see section 2.3), this would prevent any low level optimizations that are necessary to achieve maximum speed for allocation and access as well as exploitation of architecture dependent features.

4.3.1 A ReservePoolPolicy implementation for ScatterAlloc: SimpleCudaMalloc

Since ScatterAlloc is a pool based memory allocator, it requires at least one memory area that can be used to hold the data structures necessary for management as well as the user data. The policy `SimpleCudaMalloc` uses the CUDA API call `cudaMalloc()` to allocate a single, fixed size pool on the GPU. This pool can subsequently be used to create heap data structures in it and serve memory requests to the allocator from memory of this pool. When mallocMC is finalized, the policy uses `cudaFree()` to remove the pool again and return all used memory.

4.3.2 An AlignmentPolicy implementation for ScatterAlloc: Shrink

Each AlignmentPolicy serves two slightly different purposes, namely correct alignment of the whole memory pool and also correct alignment of memory requests inside the pool. The former is usually unnecessary, if the pool was aligned through the CUDA API itself, as it is done by `SimpleCudaMalloc`. In case of a misalignment[3], `Shrink::alignPool()` will truncate the beginning of the given pool and update the size of the pool to reflect the changes. This ensures that the first address in the pool is always well-aligned.
When the allocator is requesting memory from the pool, `Shrink::applyPadding` will increase the size of the request until it is the least possible multiple of the chosen alignment (i.e., for an alignment of 128 bytes, a request of 200 bytes will be padded to 256 bytes). Given that the starting address for the request is well-aligned, this method ensures that the following request can be served with a properly aligned starting address as well.
If the alignment is a multiple of the cache line size, false sharing of cache lines can be avoided entirely, since each cache line will accommodate only data from a single allocated chunk. However, such a large alignment might come at the cost of a high internal fragmentation if the needed requests are a lot smaller than the chosen alignment. Therefore, the ideal size needs to be tailored to suit the application demands, which can be done by the user at compile time (see 4.3.7).

[3] If using `SimpleCudaMalloc` as ReservePoolPolicy, this can happen by choosing an alignment bigger than the CUDA internal alignment value.

4.3.3 A DistributionPolicy implementation for ScatterAlloc: XMallocSIMD

In their paper about XMalloc [16], Huang et al. describe a technique to group allocation requests for multiple workers in a workgroup that cooperates in a SIMD fashion. The proposed algorithm was present in ScatterAlloc and is used in mallocMC as part of the DistributionPolicy **`XMallocSIMD`**, enabling support for SIMD execution as requested in section 2.3:
The function **`XMallocSIMD::collect()`** first determines, if the requests in the SIMD group are eligible for combined allocation. This is mainly limited by the size of individual requests, as no single worker may request more memory than $\frac{1}{32} \cdot$ **`pagesize`**.[4] If this precondition is fulfilled, all participating workers will combine their request sizes to be allocated by a randomly chosen master. This avoids any contention that might arise from multiple workers in the SIMD group trying to access a shared resource at the same time. Otherwise, no optimization will take place and every worker tries to allocate their own chunk.

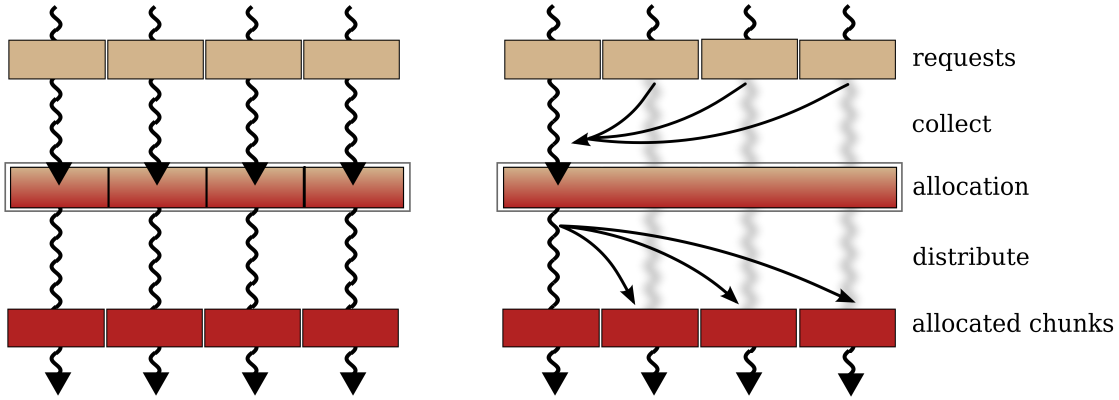


Figure 4.1: Memory allocation with 4 SIMD workers. On the left, no special SIMD optimizations are in place and each worker allocates its own chunk, similar to **`DistributionPolicy::NoOp`**. On the right, one master collects the desired allocation sizes from all participating workers, allocates a large chunk of memory and distributes it to the other workers. This behaviour is present in **`DistributionPolicy::XMallocSIMD`**.

After the allocation took place, memory must be distributed from the allocating master to the workers issuing the initial request. If no SIMD optimization was possible, this is trivially done through the identity relation. If there was a single master worker, each thread has to determine its offset inside the large allocated chunk and modify its pointer accordingly.
The illustration on the right hand side in fig. 4.1 demonstrates such an allocation for four workers.
A possible downside of this particular policy class lies in the possibility of **blowup** (see section 2.3). A set of chunks that were allocated together by **`XMallocSIMD`** is only reusable after every worker that holds a pointer to that combined chunk frees said chunk again. That means that the combined chunk is not available for reallocation in case a single worker keeps its individual chunk. This problem can be mitigated by using a different DistributionPolicy, like **`NoOp`**.

4.3.4 Another DistributionPolicy implementation for ScatterAlloc: NoOp

Since ScatterAlloc can be configured to avoid aforementioned SIMD optimizations entirely, there exists another DistributionPolicy, called **`NoOp`**[5] to emulate this behaviour, which can be seen on the left hand side in fig. 4.1. This policy class will behave similar to the non-optimized case of **`XMallocSIMD`**, not performing SIMD optimizations of any kind. It might yield small speed benefits due to less overhead in

[4] Here, 32 is the maximum number of workers cooperating in a SIMD operation on NVIDIA GPUs. If some requests are bigger than this limit, the combined request might exceed the size of a single page in memory, which is not supported due to the fragmented heap layout of ScatterAlloc.
[5] The term **NoOp** is an abbreviation of **No Op**eration.

case only a single worker in the SIMD group allocates memory. It might also allow for better memory utilization and lower external fragmentation, since the requests are smaller and can therefore fill gaps in the memory layout.

4.3.5 A CreationPolicy implementation for ScatterAlloc: Scatter

As defined in chapter 3, the actual management of the heap takes place in the CreationPolicy. In case of ScatterAlloc, this includes initial creation of the page tables and serving memory requests. The new policy class, called **`Scatter`**, adopts the code necessary for these core features directly from the reference implementation, as the whole project is based on a fork of the original repository. This is beneficial, since it already provides a fast, stable and well documented solution, that might be further improved as the authors of ScatterAlloc provide updates. Configuration of this policy class was refactored as described in section 4.3.7, thereby exposing the parameters to influence the hashing function as well as removing the parameter **`USE_COALESCING`** entirely. The latter feature is now handled by the DistributionPolicy **`XMallocSIMD`** (see section 4.3.3).
ScatterAlloc already contains a function **`initHeap`**`()` to create all necessary data structures inside the pool. To complement this, a function **`finalizeHeap`**`()` was added, which resets all the data structures that might have been altered through allocation. If used in combination with the ReservePoolPolicy **`SimpleCudaMalloc`**, this is not actually necessary, since the pool will be destroyed in any case. However, a defined state after finalization helps to avoid future problems with other implementations of ReservePoolPolicy.
As an extension to the original algorithm, Scatter is capable of determining the remaining amount of free memory. Due to the fragmented nature of the heap, the standard representation of this amount in terms of free number of bytes is problematic.

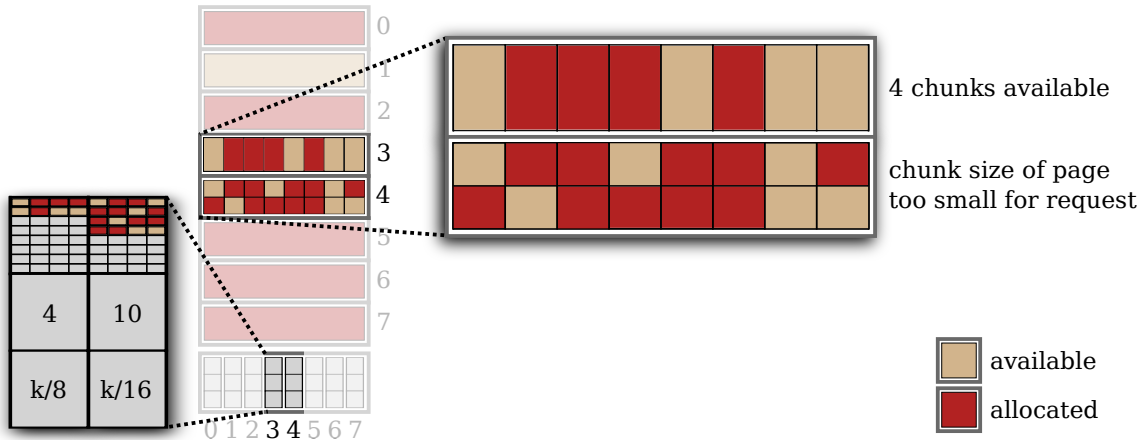


Figure 4.2: Memory layout of two pages with page size k. For a request size of $\frac{k}{8}$, only one of the pages has a usable chunk size. Therefore, only four free chunks are available, although the raw size of free memory would allow for seven chunks.

In fig. 4.2, the two observed pages of size k bytes are split into chunks of $\frac{k}{8}$ and $\frac{k}{16}$. In terms of raw size, the pages contain $7 \cdot \frac{k}{8}$ free bytes of memory. However, in case of a request size of $\frac{k}{8}$, only four of these chunks are available to satisfy the request. Therefore, the number of free bytes can be misleading. A more user friendly representation of free memory is provided by the function **`getAvailableSlots`**`(`**`size_t`**`)`, which takes a hypothetical chunk size for an allocation and determines the number of free chunks for this particular size (this type of free chunks are called *slots*).
To this end, the algorithm traverses the data structure in the same way it would in order to allocate a chunk, without actually modifying the heap. Depending on the ratio between the requested size **`slotSize`** and the configured capacity of each page **`pageSize`**, different strategies are employed: In case the request

exceeds the size of a single page, multiple pages are used for the request (see fig. 2.3). The loop iterates over the available pages sequentially and counts groups of free pages:

```
slots = 0, free = 0;
pagesNeeded = ceil(slotSize/pageSize);
for(page in pages){
  if(pageTables[page].chunkSize == 0){
    ++free;
    if(free == pagesNeeded){
      free = 0;
      ++slots;
    }
  }
  else free = 0;
}
```

Should **slotSize** be smaller than **pageSize**, the algorithm can use a different approach, where each page is considered independently:

```
for(page in pages){
  chunkSize = pageTables[page].chunkSize;
  if(chunkSize == 0) chunkSize = max(slotSize, minChunkSize);
  if(chunkSize >= slotSize) slots += countFreeChunksInPage(page, chunkSize);
}
```

In case of an empty page (**chunkSize** == 0), the page is divided as needed, but the size of one chunk may not be smaller than the minimal allowed size for a chunk. If the chunks on the given page are spacious enough to represent a free slot, the page is examined further through **countFreeChunksInPage**():

```
uint32_t hierarchyThreshold = (pageSize - 2*sizeof(uint32_t))/33;

uint32_t countFreeChunksInPage(page, chunkSize){
  if(chunkSize > hierarchyThreshold)
    chunksInPage = min(pageSize / chunkSize, 32);
  else{
    segmentSize  = chunkSize*32 + sizeof(int)
    fullSegments = pageSize / segmentSize;
    additional   = (pageSize - fullSegments*segmentSize - sizeof(int)) / chunkSize;
    chunksInPage = fullSegments + additional;
  }
  filledChunks = pageTables[page].count;
  return chunksInPage - filledChunks;
}
```

Depending on the size of the chunks that are stored inside a page, **hierarchyThreshold** is used to determine the way the page is split. If the chunks are so large, that a maximum of 32 chunks[6] suffice to fill a single page, calculation is trivial (see fig. 2.4). Otherwise, counting of used and free chunks becomes more complicated: the page is divided into up to 32 segments, each containing 32 chunks (see fig. 2.5). To use the available space more efficiently, additional chunks can be located outside of the segments (called *additional* in the code above).
Since accesses to the pages are mutually independent, they can be easily parallelized. As explained in section 4.2, the function can be called from the host or accelerator. Parallelization from the host is trivial, since the kernel call has full control over cooperation between workers. Concurrency can be freely adjusted and will improve computation time.
Calling this function from the accelerator, however, involves a number of pitfalls. To correctly decompose the domain of available pages and sum up the resulting numbers, the function needs to determine the total number of cooperating workers. This step requires synchronization between threads, which

[6] Bookkeeping is done through a bitfield consisting of an **int** variable, which is 32 bit wide on NVIDIA GPUs.

can be difficult: CUDA allows conditional execution of code, causing some workers to not even call the function in the first place. On the grid level, this is problematic since CUDA offers no way to synchronize between multiple blocks inside a kernel call.[7] This alone would limit cooperation to block boundaries. Even worse, the conditional execution inside a block can limit execution of the function to a subset of workers. The necessary synchronization within the block by using `__syncthreads()` could cause a deadlock in certain configurations. Therefore, a non-blocking approach is required. Fortunately, CUDA offers special intrinsic voting-functions, that can be used for that purpose. As a downside, this limits cooperation between workers to the width of these voting function.[8]
As a result of these limitations, a maximum of 32 workers can cooperate to determine the amount of free memory directly on the accelerator. This can lead to the unexpected behaviour that calling the function `getAvailableSlots()` from the host might actually be faster than calling it directly on the accelerator due to the increased parallelism.
As a last remark, it should be noted that the function always returns exactly the number of available slots of one size. It does not provide any guarantees about the location or fragmentation characteristics of these slots. Also, since workers analyse pages concurrently, the result is undefined if the function is called with multiple different input sizes inside the same SIMD group.
In order to comply with the interface, Scatter needs a function to determine if a memory request was actually successful. A failed request is characterized by a request size `s` greater than zero, which resulted in a pointer `p` that is `NULL`:

```
bool isOOM(){
  return s && (p == NULL);
}
```

4.3.6 A OOMPolicy implementation for ScatterAlloc: ReturnNull

To handle the cases where `CreationPolicy::isOOM()` returns `true`, the OOMPolicy `ReturnNull` offers a solution that resembles the behaviour of ScatterAlloc by simply returning a `NULL` pointer. Although the use of exceptions would allow for more advanced error handling, exceptions are unfortunately not yet supported by CUDA.[9]

4.3.7 Compile time configuration of policy classes

In its reference implementation, ScatterAlloc uses a single statement like

```
// pagesize, accessblocks, regionsize, wastefactor, use_coalescing, resetfreedpages
#define HEAPARGS 4096, 8, 16, 2, true, false
```

to configures various parameters like pagesize or the use of SIMD optimizations. To improve flexibility and ease of use, while maintaining the speed that results from compile time configuration, the new implementation relies on template metaprogramming. To configure the heap of the Scatter policy class to behave identically to the above configuration of ScatterAlloc, one would write

```
struct Config1{
  typedef boost::mpl::int_<4096>   pagesize;
  typedef boost::mpl::int_<8>      accessblocks;
  typedef boost::mpl::int_<16>     regionsize;
  typedef boost::mpl::int_<2>      wastefactor;
  typedef boost::mpl::bool_<false> resetfreedpages;
};
```

[7] *Grid* and *block* here denote the CUDA concepts of the same name.
[8] The used voting function `__ballot()` is limited to CUDA warp-width, which is currently set to 32.
[9] Using CUDA 6.0 on a GPU with compute capability 3.5.

This format introduces a number of benefits: Apart from providing the raw values, the parameters are now coupled with static types and labelled with explicit identifiers. Instead of relying on comments to describe each parameter, the code is now self-documenting, which improves readability and reduces programming errors. Furthermore, the parameters are no longer combined in a fixed preprocessor macro, but mutually independent and encapsulated into a C++ struct. Note, that the struct contains only five parameters, instead of the initial six. This is a result of the split into policies: the *use coalescing*-parameter is now expressed by choosing between different DistributionPolicies.
It also becomes possible to provide a struct containing a default configuration for each policy class that can subsequently be inherited from, to override some of its members. For the example, the default configuration struct for the heap of Scatter can be obtained through

```
mallocMC::CreationPolicies::Scatter<>::HeapProperties;
```

In case of ScatterAlloc, it is possible to ensure optimal performance for allocation, fragmentation and caching behaviour by adjusting the policy classes to fit the data patterns used in the program. Initial values can be obtained through educated guesses based on expected allocation sizes, which are then further improved through benchmarking. In these scenarios, it is desirable to change configuration parameters for each run. The configuration structs[10] can be individually overridden during compilation by passing the required arguments to the build system:

```
make -DMALLOCMC_CP_PAGESIZE=1024
```

will use the configuration structs inside the code for all parameters except **`Scatter::pagesize`**, which will be `1024`. The names of each configuration parameter of the implemented policy classes can be found in the code documentation.

[10] And, therefore, also the default values of the configuration structs.

5 Performance evaluation

This chapter offers benchmarks to compare existing memory allocators with different configurations of mallocMC. This provides comparable information about the overhead that is introduced by mallocMCs interface and design when used to wrap an existing allocator. Furthermore, performance of the implemented allocation algorithms is compared to competing solutions in terms of allocation speed and scaling for an increasing number of workers. The last section describes means that were implemented to verify the correctness of the program.
The tests were conducted on compute nodes of the *Hypnos* cluster of the *Helmholtz Zentrum Dresden Rossendorf*. Each node is equipped with a quad core Intel Xeon E5-2609 CPU (2.40GHz), 64GB RAM and 4 NVIDIA Tesla K20M GPUs. It contains 5GB GDDR5 RAM, 2496 CUDA cores and is overclocked to 758MHz as a default. The C++ code is compiled with the *GNU C++ compiler* version 4.6.2 and the CUDA framework version 6.0. The benchmarks in section 5.1 are run on a single node using just one GPU. The benchmarks in section 5.2 are run on four nodes, using four GPUs on each node.

5.1 Synthetic benchmark: Framework overhead

For all results in this section, the benchmarked kernels were preceded by another CUDA kernel call to remove the time needed to initialize the CUDA context on the GPU from the measurements. For all time-related measurements, the benchmarks will exclude setup-procedures like creation of heap structures and also parts of the kernel that are not directly responsible for allocations.
Elapsed time for the kernels is measured through the CUDA internal `clock64()` function, which offers the most precise results and ignores any overhead that might occur due to kernel launches.
The synthetic benchmark aims to demonstrate the influence of the abstraction layer provided by the policy based design of the new interface. As a baseline serves a plain version of ScatterAlloc that was enhanced by several bug-fixes for correctness. This is compared to mallocMC with policies that aim to resemble the behaviour of ScatterAlloc. Additionally, the default CUDA toolkit allocator is used to demonstrate differences to the standard solution for memory allocation on CUDA accelerators.

5.1.1 Setup

To obtain the necessary data, the benchmark is run for different allocation sizes $N_{byte} \in \{16, 32, 64, 128\}$ bytes. The values were chosen to cover the two important algorithms of allocation when using a page size of 4096 byte with ScatterAlloc. Other sizes are possible, but can be expected to yield similar results since the base algorithms do not change. For each value of N_{byte}, the executable is run with different numbers of concurrently active workers N_{worker}.[1] The values were chosen for a smooth distribution in logarithmic scale and include typical numbers of workers that achieve optimal efficiency according to the CUDA occupancy calculator [25]. The latter is important, since many codes naturally try to use similar values. Other values are possible but are not expected to produce interesting outliers. As an example, appendix B contains some plots with a higher resolution. The allocation kernel uses 45 registers, which allows for a maximum of 40 active warps per multiprocessor on the selected hardware. Each GPU is equipped with 13 multiprocessors, resulting in a maximum of 16640 concurrently active threads.
The executable is run for each possible combination of N_{worker} and N_{byte}, with both of these variables staying constant during the run. To capture possible non-deterministic delays and outliers, each run is repeated with the same parameters several times.

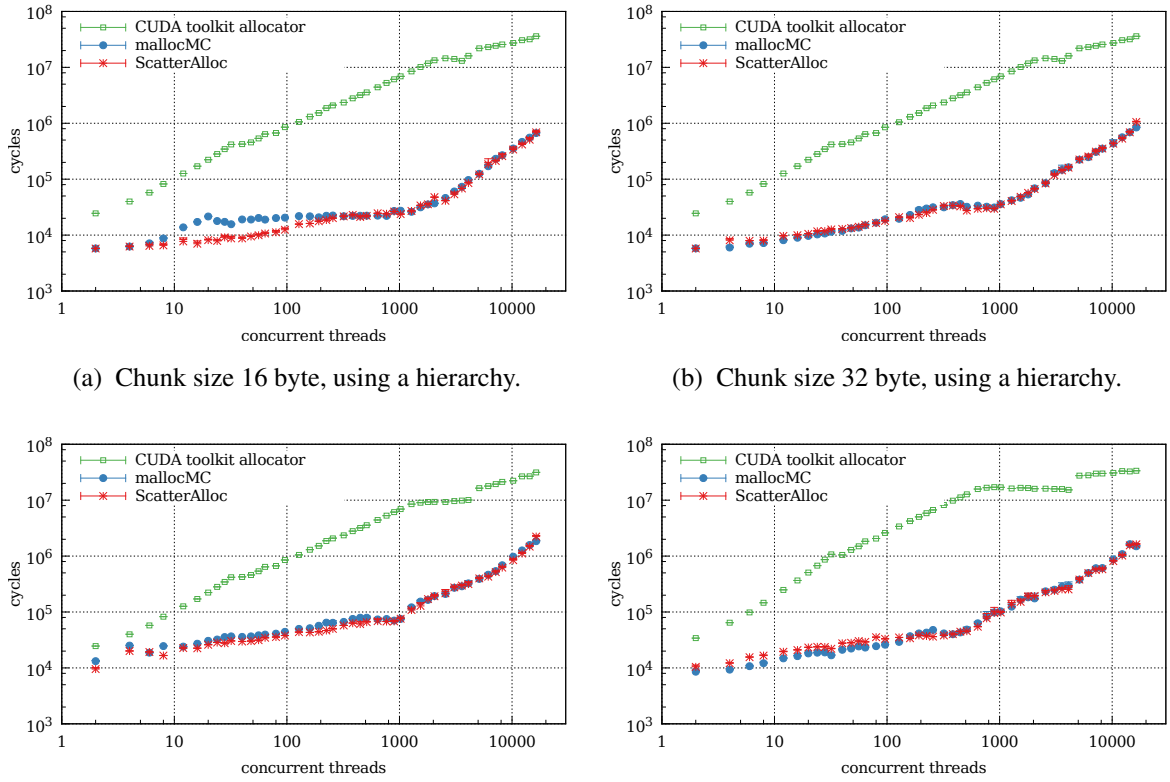
[1] $N_{worker} \in \{2, 4, 6, 8, 12, 16, 20, 24, 28, 32, 40, 48, 56, 64, 80, 96, 128, 160, 192, 224, 256, 320, 384, 448, 512, 640, 768, 896, 1024, 1280, 1536, 1792, 2048, 2560, 3072, 3584, 4096, 5120, 6144, 7168, 8192, 10240, 12288, 14336, 16384\}$

The benchmark tries to simulate a behaviour where frequent, grouped allocations and deallocations take place. Although this benchmark can only cover a fraction of the possible use cases, it is most likely sufficient for demonstrating the overhead caused by mallocMC for single allocations.
During a run of the executable, the host repeatedly creates a random number. Each time, it uses this number to launch a kernel that uses N_{worker} workers to either allocate or free a single chunk of N_{byte} byte each. Both kernels are launched with a probability of 75% based on that random number. After an initial warm-up phase, the fill level of the heap as well as the temperature of the GPU reach a constant value. Now, the number of clock cycles spent for each allocation and deallocation of memory is recorded and used to calculate an average for the run.
Both mallocMC and ScatterAlloc are configured to re-use freed pages and work without coalescing:

```
// configuration of ScatterAlloc
// pagesize, accessblocks, regionsize, wastefactor, use_coalescing, resetfreedpages
#define HEAPARGS 4096, 8, 16, 2, false, false

// configuration of mallocMC
using namespace mallocMC;

struct ScatterConfig : CreationPolicies::Scatter<>::HeapProperties{
  typedef boost::mpl::int_<4096>  pagesize;
  typedef boost::mpl::bool_<false>  resetfreedpages;
};

typedef Allocator<
  CreationPolicies::Scatter<ScatterConfig>,
  DistributionPolicies::NoOp,
  OOMPolicies::ReturnNull,
  ReservePoolPolicies::SimpleCudaMalloc,
  AlignmentPolicies::Shrink<>
> ScatterAllocator;
```

5.1.2 Discussion of results

Figure 5.1 shows the necessary number of clock cycles for a single allocation of 16, 32, 64 and 128 bytes on the vertical axis and the number of concurrently active workers on the horizontal axis. As the number of workers increases, average allocation time increases as well.
A strong agreement between mallocMC and ScatterAlloc can be observed, which indicates a high efficiency of the policy based interface, as most of the abstraction can be resolved during compile time. Assuming that an application dedicates only a fraction of its execution time to the allocation of memory, the resulting overhead on the total runtime is negligible.
Analysing the influence of increasing concurrency, the average duration for each allocation increases continuously with the number of active workers. This observation contradicts the expectations based on the benchmarks found in [29], which predicts an almost constant allocation time for ScatterAlloc even when facing a highly concurrent workload. This difference may be explained by a different setup of the benchmark, since mallocMC and ScatterAlloc both show the abnormal behaviour. As expected, the default CUDA toolkit allocator needs more clock cycles and scales worse than the other allocators.
MallocMC as well as ScatterAlloc display a varying performance for different values of N_{byte}. This behaviour is expected and can be explained by the page based structure of the heap. The page size remained fixed at 4096 bytes during the whole benchmark, while different values of N_{byte} were chosen. Based on the used CreationPolicy and the **`hierarchyThreshold`** (see section 4.3.5), the allocator uses a more complex algorithm for $N_{byte} < 122$ to reduce fragmentation. This algorithm increases the computational overhead but might reduce contention for a high number of active workers. Depending on the ratio of allocation size and concurrency, the allocation times can be shifted: For lower values of N_{byte}, a higher number of concurrent threads can be tolerated with less performance impact, since there



(a) Chunk size 16 byte, using a hierarchy.

(b) Chunk size 32 byte, using a hierarchy.

(c) Chunk size 64 byte, using a hierarchy.

(d) Chunk size 128 byte, using **no** hierarchy.

Figure 5.1: Allocation time for the different allocators and increasing number of concurrently active threads, measured in clock cycles per allocation. Each plot shows a different chunk size. Despite smaller differences, all four plots display the same pattern.

is a higher number of slots available. Therefore, the likelihood of contention is reduced and less time is required for an allocation.

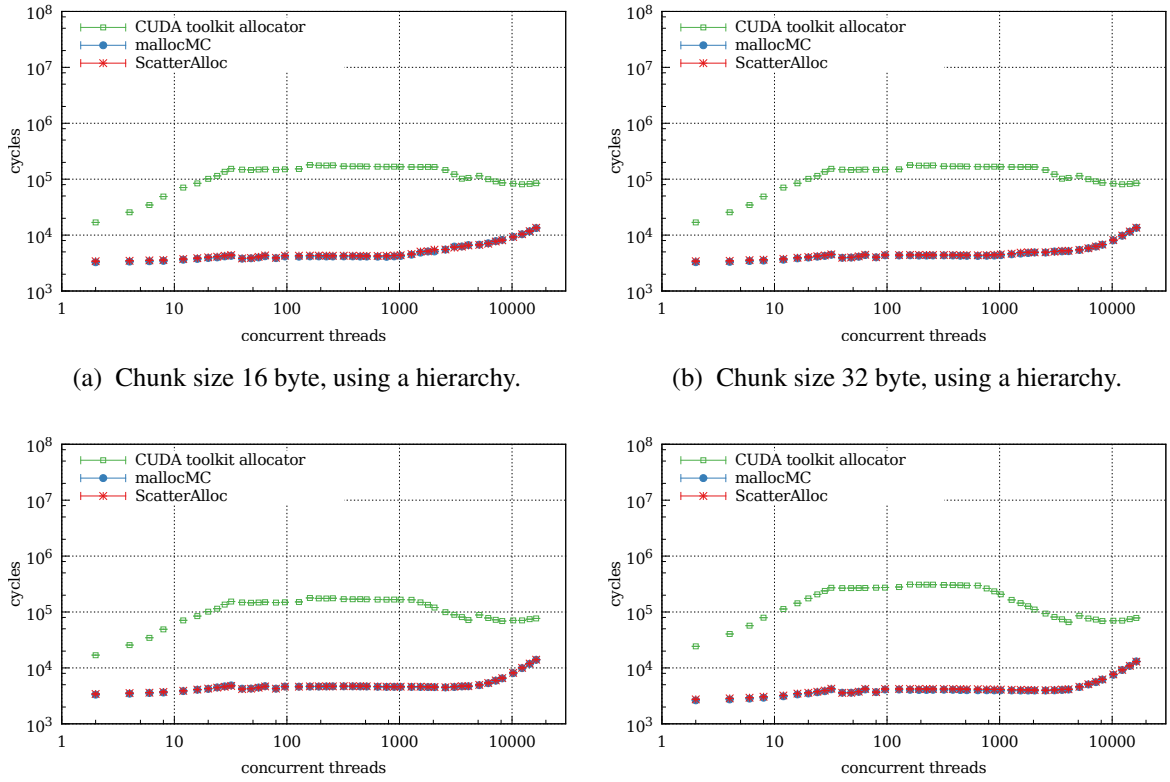
Figure 5.2 illustrates the necessary time to deallocate memory with different allocators. Similar to the allocation, the observed overhead of the interface of mallocMC is minimal. Besides that, the performance of ScatterAlloc is in strong agreement with [29] and suffers almost no increase in runtime when concurrency is increased. Again, both mallocMC and ScatterAlloc are substantially faster than the CUDA toolkit allocator.

5.2 Benchmarking the PIConGPU code

To evaluate the usefulness of mallocMC with respect to the initial motivation from chapter 1, namely to supersede the current HeapBuffer implementation used in PIConGPU, the used algorithm for mallocMC needs to be more flexible and comparably fast.

In this section, PIConGPU is used for a complete simulation to give an example of a computation with this software framework and demonstrate a typical usage pattern: During the runtime of the software, multiple time steps are processed. During each of these time steps equally sized chunks of memory are dynamically allocated and deallocated to store the state of the computation.

The data used for this benchmark was provided by René Widera from the Helmholtz Zentrum Dresden Rossendorf.



(a) Chunk size 16 byte, using a hierarchy.

(b) Chunk size 32 byte, using a hierarchy.

(c) Chunk size 64 byte, using a hierarchy.

(d) Chunk size 128 byte, using **no** hierarchy.

Figure 5.2: Deallocation time for the different allocators and increasing number of concurrently active threads, measured in clock cycles per deallocation. Each plot shows a different chunk size. Despite smaller differences, all four plots display the same pattern.

5.2.1 Setup

The selected experiment is a simulation of the *Kevin Helmholtz Instability* [5], which is run on 16 GPUs for 1600 time steps. The plotted time measurements are only taken from the first GPU (GPU 0), since all 16 accelerators are expected to yield similar results. There are two species of particles, configured to use 60% of the available global memory when the computation starts. This results in about $31.8 \cdot 10^6$ particles of each species for each GPU, using 30% of the global memory ($63.6 \cdot 10^6$ particles per GPU in total).

One run with the current HeapBuffer implementation as the allocator serves as a baseline. It is configured to use two distinct HeapBuffers, one for each species. Since these buffers are of a fixed size, they are of limited flexibility in case one of the buffers requires more memory than initially expected. This situation can occur due to particle movement within the simulation that extends beyond the domain of a single GPU. This run is then compared to a setup that uses mallocMC configured to behave like the default configuration ScatterAlloc with a few optimizations for the use case:

As discussed in section 5.1, performance of the allocator can vary with page size in relation to chunk size. In the used simulation, the allocated chunks for each species use 8192 bytes. Therefore, the experiment benefits from a page size that fits the required allocation perfectly. Furthermore, a page is sized to host exactly 32 chunks, so the non-hierarchical algorithm of ScatterAlloc is used (see section 2.4.6). `DistributionPolicies::NoOp` is chosen, since a varying number of threads will perform the allocations. Therefore, a more advanced policy like `DistributionPolicies::XMallocSIMD` would introduce a new source of fragmentation due to the differently sized combined allocations which, in turn, could impact the performance negatively. The whole necessary configuration is listed below:

```
using namespace mallocMC;

struct ScatterConfig : CreationPolicies::Scatter<>::HeapProperties{
  typedef boost::mpl::int_<8192*32>  pagesize;
};

typedef Allocator<
  CreationPolicies::Scatter<ScatterConfig>,
  DistributionPolicies::NoOp,
  OOMPolicies::ReturnNull,
  ReservePoolPolicies::SimpleCudaMalloc,
  AlignmentPolicies::Shrink<>
> ScatterAllocator;
```

5.2.2 Discussion of results

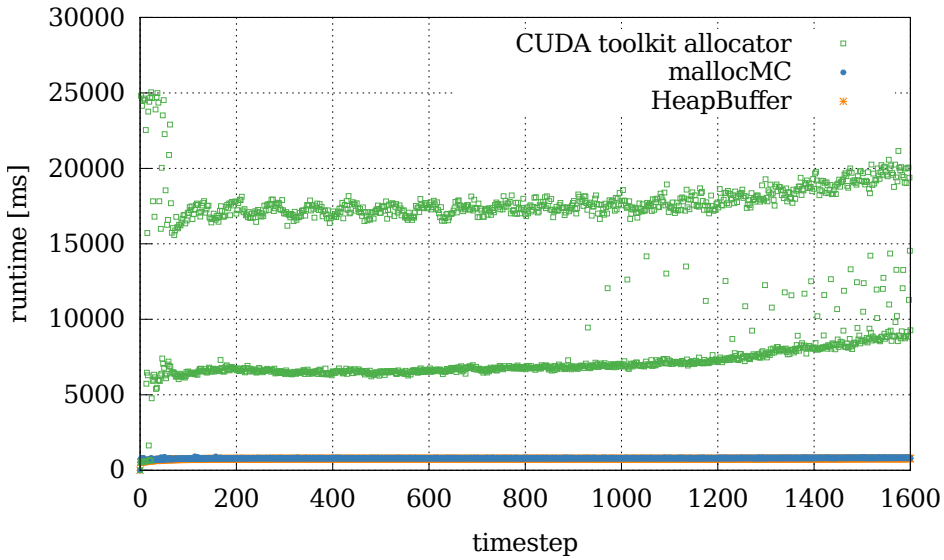


Figure 5.3: Runtime of a whole PIConGPU simulation comparing all three memory allocators. The use of the CUDA toolkit allocator slows down the simulation by orders of magnitude and shows a strong anomaly, where every second time step takes significantly longer.

The plots in fig. 5.3 illustrate the duration of each timestep for all allocators. Most notably, the simulations configured to use the HeapBuffer and mallocMC are orders of magnitude faster that the simulation that relies on the CUDA toolkit allocator. Furthermore, there is a short initial phase of fluctuating run times before the necessary duration to complete a time step stabilizes. However, even in the stable phase, about every second time step takes notably longer. Therefore, each allocator appears to have two plots in the graphic, while the duration between data points actually just alternates frequently. This pattern is pronounced most severely for the CUDA toolkit allocator, but can be observed for all tested algorithms (see fig. 5.4). As of yet, no satisfactory explanation for this phenomenon has been determined. A direct influence of the allocator appears unlikely, since all allocators show a similar behaviour, despite their entirely different approaches to manage the memory. However, there remain other possible factors, such as problems with the timer of the test system, load imbalances within the simulation as well as undocumented behaviour within PIConGPU or even the NVIDIA driver API itself.
Figure 5.4 is a plot of the same simulations, omitting the CUDA allocator to emphasize the differences between mallocMC and the current allocator. As for the comparison, both allocators show a similar characteristic, including the different durations for each second timestep. In terms of runtime, the use of

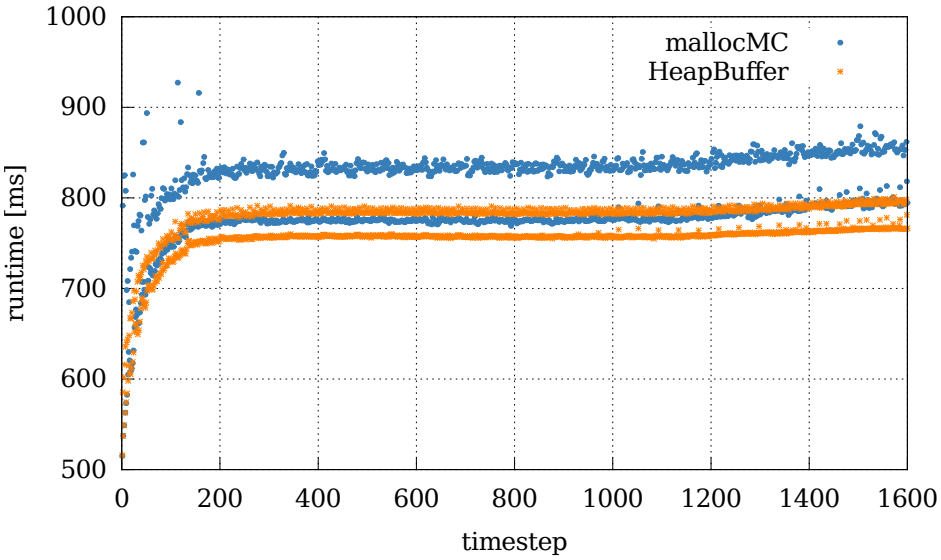


Figure 5.4: Runtime of for a whole PIConGPU simulation comparing the old and the proposed memory allocator. Although the HeapBuffer implementation runs minimally faster, both allocators show similar performance characteristics and a similar anomaly, where every second time step takes notably longer.

mallocMC impacts the overall performance of the program by about 4.6% on average, with a median of 3.7% and a standard deviation of 2.8%.

Figure 5.5 illustrates this performance difference of mallocMC normalized to the runtime of the current HeapBuffer as a histogram. The histogram shows two peaks, one around 103% and one around 107% of the runtime necessary for the simulation with the HeapBuffer. The two peaks again reflect the observed anomaly. They also indicate that the influence has a non-linear behaviour: If the runtime of a timestep is longer, the anomaly becomes overproportionally pronounced. Therefore, the linear normalization can not collapse the relative difference in run times. Note, that there are a few values with up to 36% of runtime difference that are omitted from the plot.

Although this increased runtime is undesirable, it might be negligible in the face of increased flexibility in terms of memory usage. The amount and mixture of particles present in each cell of the simulation (and therefore on each accelerator) can vary greatly over the time of an experiment. Currently, PIConGPU only supports two different species at the same time so the effects are demonstrated by a sample calculation:

The separate heaps of the HeapBuffer implementation are of a fixed size for each species. Assuming that all n species are modelled with the same number of particles, the memory is statically divided into n equally sized heaps, one for each species. Therefore, a species can never occupy more than $\frac{1}{n}$ of the available memory on each accelerator. With mallocMC in its current configuration, however, each species is capable of occupying the whole available memory for itself if necessary, allowing for the highest possible amount of particles for the simulation.

Figure 5.6 depicts this effect for up to 16 species: mallocMC in its current configuration and the CUDA toolkit allocator can distribute the memory freely at runtime, allowing 100% of the available memory to be used by a single species. The HeapBuffer on the other hand strongly limits this number. Therefore, simulations are limited to a very low number of species or a very low number of particles for each species.

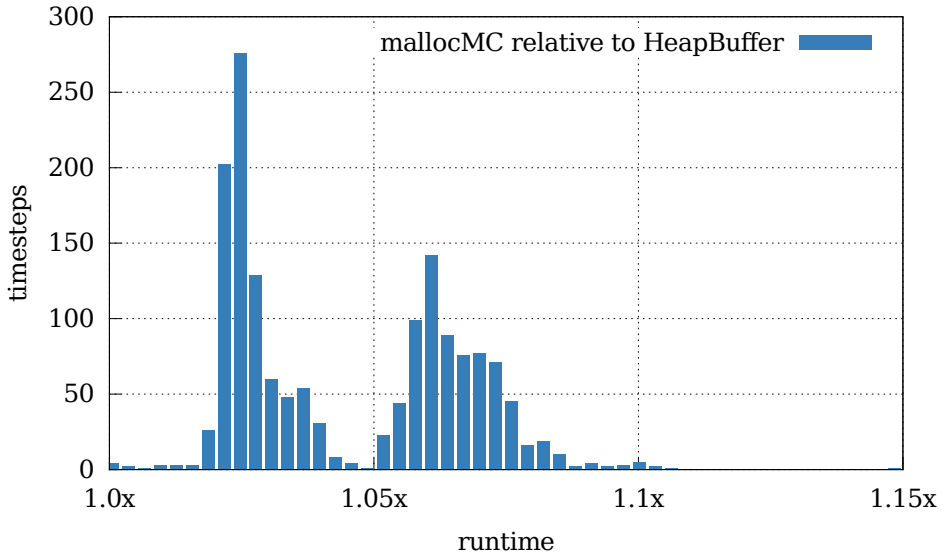


Figure 5.5: The run times of each timestep of mallocMC are normalized to the run times of the HeapBuffer and plotted as a histogram. In average, mallocMC takes 4.5% longer, with a standard deviation of 2.8% and a median of 3.7%. The two peaks of the histogram reflect the non-linearity of the performance anomaly.

5.3 Analysis and critique of achieved results

The executed benchmarks support the two most important performance goals of this work: On the one hand, the low overhead of the mallocMC interface. This allows to wrap a variety of existing memory allocators so they can be transparently interchanged and compared. On the other hand, the wrapped allocator ScatterAlloc is confirmed to be suitable as the memory allocator in PIConGPU. It achieves similar performance as the current HeapBuffer implementation, but adds the flexibility of dynamic memory allocation.

A possible critique arises from the fact that each benchmark used only a limited number of chunk sizes. This pattern clearly favours a slab-based allocator like ScatterAlloc and ignores certain benefits of the CUDA toolkit allocator, which can achieve rather consistent allocation times for most sizes. However, many of the current codes on accelerator hardware rely on this memory layout with fixed sizes. Therefore, the benchmark remains representative for the considered scenarios. PIConGPU in particular uses only a very limited number of different chunk sizes for a given simulation, as seen in section 5.2. As a consequence, slab-based allocators are not only very well suited for the workload in general, but can even be fine-tuned to the specific chunk sizes, which increases allocation speed even further.

Lastly, the periodic changes in runtime remain without explanation. Further investigation will be necessary to rule out errors when measuring the runtime for each step as well as any possible problems in PIConGPU.

5.4 Verification of correctness

To ensure not only the speed of the algorithm but also its correctness, it is important to first identify the possible problems. For ScatterAlloc in particular, there are several expected sources of errors that need to be taken into account.

Since all memory is already pre-allocated from the accelerator's memory to form the heap, the requested chunks are served in form of pointer addresses into this heap. Therefore, errors in address calculations are possible that lead to overlapping memory chunks. In such a case, different workers might hold data

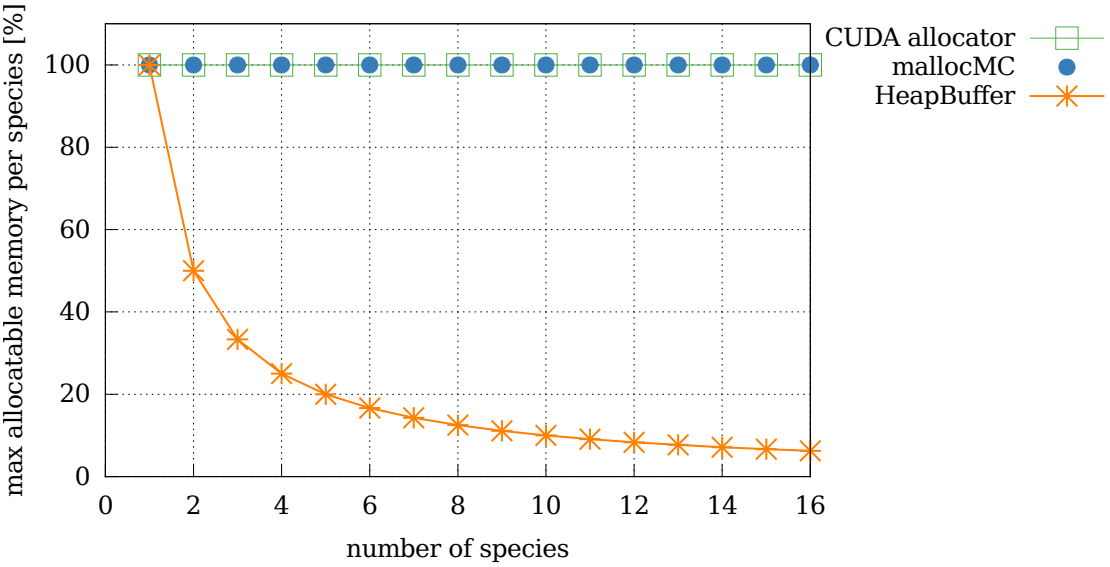


Figure 5.6: Maximum number of particles per species on a single GPU. The CUDA toolkit allocator as well as mallocMC can distribute memory to a single species dynamically. The HeapBuffer implementation is limited by the initial, static split of the memory into $\frac{1}{n}$ separate heaps. The data is based on a sample calculation

on the same physical memory. When one of the workers writes data, it will corrupt the data of the other workers on that location.

Another problem arises from ScatterAlloc's separation of the heap into accessblocks and pages. If the algorithm misses free areas in its heap, some form of memory loss might occur. This memory loss can be constant, meaning that some pages are not considered during the entire runtime of the program[2], or emerge while running. The latter is possible if pages are not freed properly or the metadata about pages reaches an inconsistent state. This may not be confused with **blowup** (see section 2.3), which is caused by the algorithm itself rather than a programming error.

To verify the correctness in terms of memory corruption, a simple program was developed to repeatedly write and read memory locations to verify that no other worker did access the chunk: mallocMC is configured to use most of the available global memory of the GPU. Then, a number of workers is used to request chunks of memory and write data to it with a pre-defined stride. The written data is a checksum generated by the worker's ID and a counter exclusive for the worker. After the heap is completely filled, the counters are reset and the workers are executed again with the same striding pattern. Now, the checksum is generated again, but instead of writing, it is compared to the already stored checksum at the current location. If another worker was assigned an overlapping address, it would have written its own checksum, thereby corrupting the existing checksum.

A potential loss of memory can now be determined more easily by utilizing the functionality of `getAvailableSlots()`, knowledge about the heap configuration, and the intended number of chunks. During the benchmarking runs in section 5.1, the number of available slots for the benchmarked chunk sizes were determined before the time measurement and after freeing the entire heap once the time measurement was completed. In case of a completely freed heap, there number of available slots must match.[3] In some scenarios, this revealed problems with memory loss due to race conditions when updating the metadata of pages in ScatterAlloc.

[2] Before the work on mallocMC was started, René Widera encountered such an error in the reference implementation of ScatterAlloc, which lead to only $N - 1$ out of N possible accessblocks being used [27].

[3] If the used chunk size of completely freed pages is not reset, ScatterAlloc considers the page only as available, if the size of the new request matches the old one. Therefore, the numbers only match reliably when using the option to reset freed pages.

The developed bug-fixes were merged into the ScatterAlloc-related policies of mallocMC [7]. Furthermore, the bug-fixes as well as `getAvailableSlots()` were backported to ScatterAlloc and merged into the code branch of the Computational Radiation Physics group at the Helmholtz Zentrum Dresden Rossendorf [8].

6 Conclusion and future work

This work used the existing, massively parallel memory allocator ScatterAlloc, enhanced it through several bug-fixes, strongly improved configurability, and added functionality to examine the current fill levels of the heap. To make it a viable candidate as the future allocator in the project PIConGPU, it was adapted to the newly developed interface mallocMC. MallocMC is based on a modular, policy based design using C++ template metaprogramming and offers a way to abstract the actual allocator from the program logic. This abstraction can be resolved during compile time to eliminate almost all performance penalties. It will also allow to transparently change or modify the underlying allocation algorithm in the future, without modifying PIConGPU itself.
Synthetic benchmarks showed only minimal impact on performance of this abstraction and confirmed ScatterAlloc's performance benefits over the default allocator in the CUDA toolkit. The performance of the adapted version of ScatterAlloc was shown to perform almost as fast as the currently used allocator in PIConGPU. Apart from that, the new solution offers highly increased flexibility when allocating memory, which will enable multi-species simulations with more than two species of particles.
Besides the initial deployment in PIConGPU, mallocMC is suitable for all projects and applications that utilize accelerator hardware. Nowadays, most complex programs use some form of dynamic memory allocation, although the performance penalty of this operation still poses a bottleneck. The benefit of mallocMC in such a scenario lies in the fixed interface, which allows to switch between multiple allocation back-ends. This creates the possibility to use the best available allocator for each use-case. The policy based design further supports this concept by splitting orthogonal functionalities into separate policies: upcoming algorithmic innovations can be added to an existing application simply through the creation of a new policy class. This flexibility enables an application to use always the most advanced allocation algorithm available and improves the portability to future hardware architectures, as only the relevant policy classes need to be changed.
Future work should address other desirable features for ScatterAlloc, like swapping of GPU global memory to host memory or transparently extending memory pools. To support changes in the HPC environment, ScatterAlloc itself could be ported to different platforms like Intel Xeon Phi or support different frameworks like OpenCL.
Finally, different allocators could be adapted as policies for mallocMC. This offers not only the possibility to compare the influence of these allocators on the performance of PIConGPU, but will also augment the list of currently available policy classes for mallocMC. A high number of policy classes will further improve the usability of the interface and ensure the necessary variety of algorithmic options, so that a programmer can always choose the most suitable implementation.

Bibliography

[1] Andrei Alexandrescu. The Design Is In The Code – Enhanced Reuse Techniques in C++. In *eXtreme Programming and Flexible Processes in Software Engineering Conference*, Cagliari, Sardinia, Italy, June 2000.

[2] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, November 2000.

[4] Heiko Burau, René Widera, Wolfgang Hönig, Guido Juckeland, Alexander Debus, Thomas Kluge, Ulrich Schramm, Thomas E. Cowan, Roland Sauerbrey, and Michael Bussmann. PIConGPU : A Fully Relativistic Particle-in-Cell Code for a GPU Cluster. *Plasma Science, IEEE Transactions on*, 38(10):2831, October 2010.

[5] Michael Bussmann, Heiko Burau, Thomas E. Cowan, Alexander Debus, Axel Huebl, Guido Juckeland, Thomas Kluge, Wolfgang E. Nagel, Richard Pausch, Felix Schmitt, Ulrich Schramm, Joseph Schuchart, and René Widera. radiative signatures of the relativistic kelvin-helmholtz instability. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, sc '13, pages 5:1–5:12, new york, ny, usa, 2013. acm.

[6] c-plusplus.net Community Forum. Einführung in die Policies/das Strategy Pattern. *http://www.c-plusplus.net/forum/110355*, May 2005. Accessed 2014-09-22.

[7] Computational Radiation Physics group. github code repository of mallocMC. *https://github.com/ComputationalRadiationPhysics/mallocMC*, 2014. Accessed 2014-08-30.

[8] Computational Radiation Physics group. github code repository of ScatterAlloc. *https://github.com/ComputationalRadiationPhysics/ScatterAlloc*, 2014. Accessed 2014-08-30.

[9] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[10] Intel Corporation. Excerpts from a conversation with Gordon Moore: Moore's Law. Video Transcript, 2005.

[11] William J. Dally. The future of GPU computing. In *the 22nd Annual Supercomputing Conference*, Portland, OR, USA, 2009.

[12] Dave Dice and Alex Garthwaite. Mostly Lock-free Malloc. *SIGPLAN Not.*, 38(2 supplement):163–174, June 2002.

[13] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, Sept 1972.

[14] Wolfram Gloger. Dynamic memory allocator implementations in Linux system libraries. *http://www.malloc.de/en*, June 2006. Accessed 2014-05-26.

[15] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[16] Xiaohuang Huang, Christopher I. Rodrigues, Stephen Jones, Ian Buck, and Wen-Mei Hwu. Scalable SIMD-parallel Memory Allocation for Many-core Machines. *J. Supercomput.*, 64(3):1008–1020, June 2013.

[17] Xiaohuang Huang, Christopher I. Rodrigues, Stephen Jones, Ian Buck, and Wen mei Hwu. XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines. *Computer and Information Technology, International Conference on*, 0:1134–1139, 2010.

[18] Intel. Intel® Many Integrated Core Architecture (Intel® MIC Architecture). *http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html*, 2013. Accessed 2014-05-20.

[19] Jaakko Järvi. Tuples and multiple return values in C++. Technical Report 249, Turku Centre of Computer Science, March 1999. http://www.tucs.fi.

[20] Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 179–188, New York, NY, USA, 1995. ACM.

[21] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, September 2011.

[22] Doug Lea. A memory allocator. *http://g.oswego.edu/dl/html/malloc.html*, April 2000. Accessed 2014-05-26.

[23] NVIDIA DevTalk Forum members. Overwriting device-side malloc(). *http://devtalk.nvidia.com/default/topic/720863*, April 2014. Accessed 2014-04-28.

[24] NVIDIA. CUDA C Programming Guide. *http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html*, 2013. Accessed 2014-05-24.

[25] Cliff Woolley NVIDIA. CUDA Occupancy Calculator. *http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls*, 2012. Accessed 2014-08-29.

[26] Jan Reher. Policy-Based Design in the Real World. *Dr. Dobb's Journal*, October 2004.

[27] René Widera. fix #2 accessBlog bug - ccb2d0ab72ba7426a0 - ComputationalRadiationPhysics. *https://github.com/ComputationalRadiationPhysics/scatteralloc/commit/ccb2d0ab72ba7426a0*, January 2014. Accessed 2014-08-27.

[28] Computational Radiation Physics Group Helmholtz Zentrum Dresden Rossendorf. PIConGPU – A Particle-in-Cell Code for GPGPUs. *https://github.com/ComputationalRadiationPhysics/picongpu*, 2014. Accessed 2014-05-22.

[29] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU. In *Proceedings of Innovative Parallel Computing (InPar12)*, 2012.

[30] Bjarne Stroustrup. *The C++ Programming Language, 4th Edition*. Addison-Wesley Professional, 4th edition, May 2013.

[31] Herb Sutter. The free lunch is over – a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, March 2005.

[32] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall International, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[33] Sven Widmer, Dominik Wodniok, Nicolas Weber, and Michael Goesele. Fast Dynamic Memory Allocator for Massively Parallel Architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 120–126, New York, NY, USA, 2013. ACM.

A Usage tutorial

This chapter will give a brief introduction on how to configure and use mallocMC in a basic setup. The aim is to use mallocMC to wrap ScatterAlloc as a drop-in replacement that substitutes the default allocator in existing NVIDIA CUDA code.

The includes for mallocMC are split into different files (see listing A.1): The host class is always required, while the overrides are only needed to hide the global object and improve the user experience. The base installation also provides a single include file for each policy.

```
1  // Load the hostclass
2  #include "mallocMC/mallocMC_hostclass.hpp"
3  #include "mallocMC/mallocMC_overwrites.hpp"
4
5  // Load all available policies for mallocMC
6  #include "src/include/mallocMC/CreationPolicies.hpp"
7  #include "src/include/mallocMC/DistributionPolicies.hpp"
8  #include "src/include/mallocMC/OOMPolicies.hpp"
9  #include "src/include/mallocMC/ReservePoolPolicies.hpp"
10 #include "src/include/mallocMC/AlignmentPolicies.hpp"
```

Listing A.1: Include files for mallocMC.

Next, the intended policies need to be configured (listing A.2). The available parameters for each policy class can be found in its documentation. In this case, all configuration structs rely on **`boost::mpl`** to supply their parameters wrapped into types. Note, that **`ScatterConfig`** actually inherits from a default provided by the policy class and only overwrites certain values.

```
11 // configure the CreationPolicy "Scatter"
12 struct ScatterConfig : mallocMC::CreationPolicies::Scatter<>::HeapProperties{
13   typedef boost::mpl::int_<4096>  pagesize;
14   typedef boost::mpl::bool_<false> resetfreedpages;
15 };
16
17 // configure the DistributionPolicy "XMallocSIMD"
18 struct DistributionConfig{
19   typedef ScatterConfig::pagesize pagesize;
20 };
21
22 // configure the AlignmentPolicy "Shrink"
23 struct AlignmentConfig{
24   typedef boost::mpl::int_<16> dataAlignment;
25 };
```

Listing A.2: Configuration structs with parameters suitable for the intendend policy classes.

Based on the created configuration structs, the actual allocator can now be composed as a C++ typedef. The class **`mallocMC::Allocator`** is the host class that uses the policy classes as template parameters. The policy classes themselves use the configuration structs as template parameters (listing A.3).

```
26 using namespace mallocMC;
27 // Define a new allocator and call it ScatterAllocator
28 // which resembles the behaviour of ScatterAlloc
29 typedef Allocator<
30   CreationPolicies::Scatter<ScatterConfig>,
31   DistributionPolicies::XMallocSIMD<DistributionConfig>,
32   OOMPolicies::ReturnNull,
33   ReservePoolPolicies::SimpleCudaMalloc,
34   AlignmentPolicies::Shrink<AlignmentConfig>
35 >ScatterAllocator;
```

Listing A.3: The allocator is composed by creating a new type based on the host class.

In order to improve usability, listing A.4 shows how to set the newly defined allocator type to be used by the following functions to create a global object of this type. The macro on line 40 is used to override the default functions **`malloc`**`()` and **`free`**`()` of the accelerator. On NVIDIA CUDA GPUs, this will actually also replace calls to **`new`** and **`delete`**, since both internally use the same functions. Due to the way a CUDA program is built by the compiler, a function may not override itself. Therefore, this macro can not be used when the internal allocator of mallocMC utilizes the original CUDA device functions **`malloc`**`()` or **`free`**`()`. Since the used CreationPolicy **`Scatter`** does internally not rely on these built-in functions, the macro is usable here.

```
36 // use "ScatterAllocator" as the type of the global object
37 MALLOCMC_SET_ALLOCATOR_TYPE(ScatterAllocator)
38
39 // replace all standard malloc()-calls on the device by mallocMC calls
40 MALLOCMC_OVERWRITE_MALLOC()
```

Listing A.4: Macros to make the allocator usable as a drop-in replacement.

On the host, the heap of the allocator needs to be initialized before it is usable (listing A.5, line 45). After the heap is initialized, the user is able to query the number of available slots from the heap, provided that the functionality exists in the selected CreationPolicy (see section 3.6). The code in line 49 can be used to guard against empty implementations of **`getAvailableSlots`**`()` when changing the configuration of mallocMC. At the end of the program, the heap should be finalized. Therefore, the required modifications in the host code are limited to heap initialization and finalization.

```
41 __global__ void exampleKernel();
42
43 int main(){
44   // Create a heap with 1GB of free space
45   initHeap(1U*1024U*1024U*1024U);
46
47   // Test, if the used configuration offers "getAvailableSlots"
48   // and query the number of free slots with 128 byte size
49   if(providesAvailableSlots())
50     unsigned nSlots = getAvailableSlots(128);
51
52   // start the kernel
53   exampleKernel<<<...,...>>>();
54
55   // destroy the heap and free the used memory
56   finalizeHeap();
57   return 0;
58 }
```

Listing A.5: Host code to initialize the heap check for available slots and finalize the heap after kernel execution.

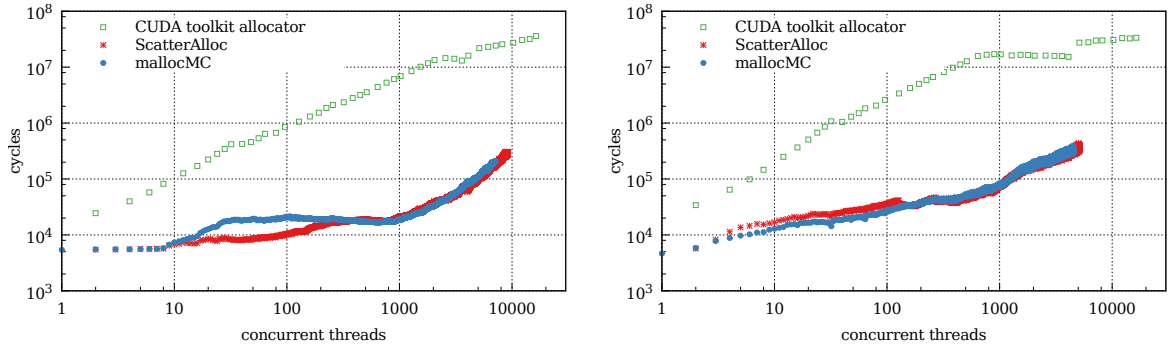
In the accelerator code itself, there might be no changes required at all, if the standard functions were overridden. Listing A.6 shows a kernel that also uses the function **`getAvailableSlots()`** to check for available slots directly on the accelerator.

```
__global__ void exampleKernel(){
  // request a chunk of memory from the heap
  int* p = malloc(128);

  if(providesAvailableSlots())
    unsigned nSlots = getAvailableSlots(128);

  ...

  free(p);
}
```

Listing A.6: Accelerator code that transparently allocates memory with mallocMC.

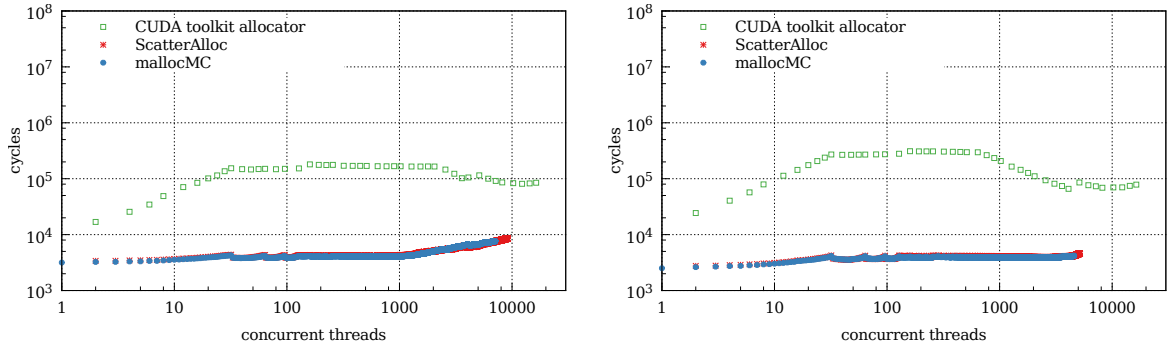
B Benchmark data

In order to demonstrate that the chosen numbers of workers in section 5.1.1 are representative and do not omit unexpected outliers, fig. B.1 shows plots of mallocMC and ScatterAlloc that were created using $N_{worker} \in \{ i \mid 1 \leq i \leq 16640 \}$. Both allocation and deallocation match the low-resolution plots. For N_{byte}, the values 16 and 128 were chosen to demonstrate cases with and without hierarchical page layouts.



(a) Allocation with chunk size 16 byte, using a hierarchy.

(b) Allocation with chunk size 128 byte, using **no** hierarchy.



(c) Free with chunk size 16 byte, using a hierarchy.

(d) Free with chunk size 128 byte, using **no** hierarchy.

Figure B.1: Allocation and deallocation times for the different allocators and increasing number of concurrently active threads, measured in clock cycles per allocation or deallocation. See section 5.1 for details about the benchmark.

Acknowledgments

I would like to thank the computational radiation physics group of the Helmholtz Zentrum Dresden Rossendorf for constant support in questions about C++, CUDA and parallel debugging. Especially René Widera, who also added mallocMC to PIConGPU to conduct the PIConGPU benchmarks; Erik Zenker, for his constructive criticism regarding the illustrations; and finally M. Steinberger et al. from Graz University of Technology for providing their original implementation of ScatterAlloc.

Copyright Information