A comprehensive study of Dynamic Memory Management in OpenCL kernels

Master thesis report

Roy Spliet (1318977, R.Spliet@student.tudelft.nl)

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science
Department of Software and Computer Technology
Parallel and Distributed Systems Group

June 5th, 2013

Abstract

Traditional (sequential) applications use malloc for a variety of dynamic data structures, like linked lists or trees. GPGPU is gaining attention and popularity because its massively-parallel architecture allows for great speed improvement for programs that can be parallelised and implemented for a platform like OpenCL. Programmers who try to port their existing sequential or even parallel program to OpenCL however will soon discover that this standard defines a subset of C with several limitations, one of which is the absence of a *malloc()* routine that can be called from an OpenCL kernel.

This document describes the results of research towards the impact of this limitation by trying to answer the question: "How should a kernel-side heap allocator be implemented in OpenCL?". To give an answer to this question, two important aspects are investigated. Firstly we seek the impact from the programmers perspective by trying to understand the use-cases for which a parallel program requires an in-kernel heap allocator. We sought for a wide variety of implementations of parallel algorithms and investigated their memory usage patterns. Secondly we aim to explain the complexity of a dynamic memory allocator by taking a closer look at the limitations imposed by hardware and the OpenCL standard, and trying to find a way to implement a memory allocator in OpenCL. Based on our findings we present KMA, the first dynamic memory allocator for OpenCL kernels.

Results from our use-case survey show that there are several cases where a programmer could benefit from an in-kernel malloc routine. Non-regular data-structures are required both for scheduling purposes and for holding intermediate results, and a memory allocator aids in creating these structures. Programmers usually find their way around the lack of a malloc routine, sometimes by creating something on their own that resembles part of one. Although a malloc routine thus is not a hard requirement, it can make the life of programmers a lot easier in these cases.

Designing and implementing a memory allocator for OpenCL is far from trivial. Its capabilities are limited by the available hardware and language constraints: hardware does not allow for communication between the OpenCL kernel and the host system, and the OpenCL specification lacks fundamental features like mutex locks. Because of these constraints, even the data-structures we might take for granted, like full-featured doubly-linked-lists, cannot be implemented in OpenCL.

We believe a memory allocator should be implemented in two layers. The bottom layer should implement behaviour like the *malloc()* routine found in C. On top of that data types can be implemented which may use datatype-specific restrictions to optimise the usage of the low-level memory allocator. Our KMA prototype shows that by carefully analysing the limitations of the OpenCL environment it is possible to implement an in-kernel memory allocator, be it limited by the constraints imposed by the GPU platform and OpenCL. Although using KMA comes with a performance penalty, it can offer the flexibility and code readability for the use-cases we identified, and offers the right level of support for optimisations if the use-case permits.

Supervisors

Dr. A. L. Varbanescu	Parallel and Distributed Systems, EEMCS	TU Delft
Dr. L. W. Howes		Advanced Micro Devices
Dr. B. R. Gaster		Qualcomm

Committee

Prof. Dr. H. J. Sips	Parallel and Distributed Systems, EEMCS	TU Delft
Prof. Dr. K. G. Langendoen	Enbedded Systems, EEMCS	TU Delft
Dr. A. L. Varbanescu	Parallel and Distributed Systems, EEMCS	TU Delft

Foreword

In the scientific world there is a long history of multiple-core processors, ranging from single-instruction-multiple-data (SIMD) devices to multiple full-featured processing units. As sequential processors were reaching its performance limits both logically and physically, these multi-core devices started to become available in the standard computer. As a consequence, programmers were forced to (re)write their programs to utilise all available parallel computational power in a device. This meant tackling a lot of problems related to synchronisation and data sharing.

Over time, the design of GPUs has grown towards that of multi-core CPUs. Both feature a number of cores that could be programmed to execute arbitrary algorithms, and programmers encounter the same problems when doing so. The biggest difference between an CPU and a GPU is the cores; where CPUs usually have a few complex fast cores, GPUs can have thousand of simpler cores. When processing a large data set of independent values, like pixels in an image, a GPU could far outperform a CPU.

Platforms like OpenCL were proposed to make this massive computational power available to any programmer. What makes OpenCL different from the other platforms available is that its functionality does not stop at the GPU: it targets a lot of devices ranging from GPUs to CPUs to FPGAs. This gives programmers the opportunity to break their heads once on writing one parallel implementation of the required algorithm, and be rewarded with acceleration on whichever device is available on the users computer without having to worry a lot about the nature of this device.

A much heard disadvantage of OpenCL is that it is relatively difficult to program in, partially because its API is limited compared to NVIDIAs Cuda. One of the missing features is a heap allocator, which C developers probably know best as *malloc*(). Having no alternative at hand, developers are currently left to their own devices. My challenge was to see what was required to create a heap allocator and see if I could implement a prototype.

This project was proposed to me by my supervisor, Ana Varbanescu, and I was immediately quite enthusiastic. Acquaintances know I have a weak spot for low-level design details of hardware. Modern GPUs tend to expose a lot of these details. With my involvement in a project that aims to write a full-featured open-source graphics driver based on reverse engineering work, my interest was already raised before even getting into the details of the project.

In my opinion, OpenCL is a platform that has a potential greater than any competing platform in the market. Besides the portability advantage, OpenCL is an open standard that is managed by an organisation anyone could join. Having seen the quality of open projects like Linux, Firefox and LibreOffice, I personally tend to favour a more democratic way of development over the closed procedures that traditional companies followed. Research to a fairly low-level problem on a platform of interest was a perfect match for me.

I therefore feel honoured to have the opportunity to contribute a small piece of work that could support any OpenCL developer in their task of writing maintainable and portable code. I could not have done this though without extensive feedback (and an occasional push forward) from my supervisor Ana Varbanescu and the contributions of Lee Howes and Ben Gaster, all of who helped me think in different ways about problems I encountered. Given my stubbornness I definitely owe them my gratitude. Thanks also go out to the friends and room mates that listened to my complains, to my parents that carried the financial burden of my study, and to anyone else who made this project possible.

I hope this work proves to be of value to its readers, and look forward to any discussion that might follow from it.

Roy Spliet

Contents

Chapter 1

Introduction

An important tool in the hands of any C developer is the heap allocator, also referred to as (dynamic) memory allocator, heap manager or malloc-routine. This heap allocator complements the static array by having means of reserving a chunk of memory in runtime rather than on loading an executable. This flexibility makes it more efficient to work with data that is sent through a socket, read from a file, or otherwise needs to be transformed to a more redundant form like a search tree.

Internally a memory allocator is a piece of software that delegates and administrates any memory request the user program might have. In the case of C, it is part of the POSIX standard[1] as implemented by libraries like glibc[25]. Each user program has its own heap that maps memory from addresses in its own virtual memory range.

Heap allocators successfully utilise caching mechanisms to work with the rather big chunks of memory (pages) the operating system provides and to improve performance. They ensure memory space is available to the program upon request, and is handed back to the operating system when no longer required. This means the algorithms that manage the state of the heap ideally minimise the execution time and memory usage, but generally have to make a trade-off between the two[29].

With the recent rise of parallel systems, programmers were facing new challenges in writing efficient code. For sequential execution there is a fairly linear correlation between the number of instructions executed and the physical execution time (wall-time). For parallel execution on the other hand, extra measures have to be taken in the code to make a program "thread-safe", meaning the state of the entire program is guaranteed to remain correct even if multiple threads are running in parallel. Handling data dependencies and synchronisation between the different cores are some measures to ensure thread-safety. However, these measures introduce idle time in the execution of a program, increasing the wall-time a program takes to run.

Increasing the performance of a program now means minimising the idle-time necessary for thread-safety as well as writing code that executes as little instructions as possible. Of course this is also true for designing a thread-safe heap allocator for parallel systems. A lot of research was done on how to efficiently design and maintain the state of the heap when accessed by multiple threads at the same time[16, 31, 7, 23].

A popular subset of parallel processing devices nowadays are Graphics Processing Units, or GPUs. Being designed to work on large amounts of vectors for graphical processing, these devices implicitly have a great parallel processing capacity. With the introduction of General Purpose GPUs(GPGPUs) and accompanying (de-facto) standards, such as Cuda[36] and OpenCL[27], this processing capability became accessible to non-graphics related applications.

Although heap allocators are much studied and frequently used, the OpenCL standard currently does not contain a heap allocator. Other parallel platforms like Cuda[36] and OpenMP do give the programmer access to a heap and an accompanying *malloc()*-routine, which raises some obvious questions about this difference.

1.1 Problem description

The goal of our research is to understand what place a heap allocator would take in the OpenCL platform. More specifically, we wish to answer the following question:

How should a kernel-side heap allocator be implemented in OpenCL?

To give an answer to this question, we approach this problem like any other software development project by following the following steps:

1. Derive the (users) requirements
2. Design the heap allocator
3. Implement and debug the heap allocator
4. Gather feedback and repeat any steps necessary

Deriving the users requirements is done by means of a use-case survey. Looking at the problems users are trying to tackle with parallel systems and their approach in their implementation gives a good insight in the position a heap allocator could take in an OpenCL-enabled program. Of course, this means investigating a series of programs implemented in platforms that do have a heap allocator, but also trying to understand how this allocator is used in sequential programs.

From these results we not only get a motivation for building a heap allocator, but also derive some more insight on the demands for such an allocator. Based on these demands and the specifics of OpenCL we then design and implement a heap allocator we call Kernel Memory Allocator, or KMA. Due to the complexity of OpenCL and parallel systems in general this is an iterative process: bugs and problems in an implementation lead to more insight in the algorithms we use and in the available OpenCL platforms, in turn affecting the constraints and design of the final heap allocator. The design of KMA is inspired by earlier attempts at creating a dynamic heap allocator made in literature[29, 7, 23].

1.2 Outline of this document

In this document we present KMA, the Kernel Memory Allocator, as a proof-of-concept to show the possibilities for dynamic memory allocation within the boundaries of an OpenCL environment. We explain the design of our prototype and evaluate its performance by looking at both its memory wastage and its performance. Based on these results we discuss the benefits, costs and other limitations that KMA has, proposing several improvements for a possible KMA-2.

A more in-depth explanation of memory allocators, the OpenCL platform and some of its differences from CPU execution is given in chapter 2. The use-case research and results of this study can be found in chapter 3. The design details of the memory allocator are explained in chapter 4. Implementation details are given in chapter 5, and chapter 6 will contain an analysis of the proposed algorithm and benchmark results summarising the implementation of this memory allocator. After showing results, we discuss the current state and the future of KMA in chapter 7.

Chapter 2

Background and related work

2.1 Parallel programming: locking and lock-free algorithms

When talking about parallel computation, there are several models that could be referred to. In our work we consider three. Multiple-Instruction Multiple-Data or MIMD is the most commonly seen model on processors. In this model there are several fully functional cores, each executing its own code. The opposite of MIMD is Single-Instruction Multiple-Data or SIMD. Instead of having multiple fully functional independent cores, in the SIMD model a single instruction on a single core leads to the same computation on different data elements. Finally there is the Single-Instruction Multiple-Threads (SIMT) model. A thread is a running instance of a program. Multiple instances that share variables and other resources can be running in parallel, in which we speak of "multi-threaded execution". The SIMT model differs from the SIMD model in the way they are programmed: where in the SIMD model code contains explicit instructions to work on a vector of data, in the SIMT model instructions work on scalar data and are vectorised by hardware by launching multiple threads. In the SIMT model cores are often clustered and thus execute the same instruction on multiple cores at the same time, typically with different data elements. This last model is used on many modern graphics cards. In this document we will assume the MIMD or SIMT model unless otherwise specified.

In the introduction we briefly discussed the differences between sequential and parallel programming. The major challenge that any programmer will face when designing or implementing a parallel algorithm is ensuring that all shared values remain consistent. Without considering parallel access to shared values, a program will most likely encounter "race conditions" that occur when two (or more) threads try to alter the same shared variable at the same time[19]. After both threads finish their write operation the outcome can be either of the two values, thus the value of this shared variable is unpredictable. Depending on the algorithm this may lead to a corrupt state of the shared data or unpredictable behaviour.

Algorithms that are designed to avoid such race conditions can be classified into two categories: "locking"-algorithms and "lock-free"-algorithms.

Locking algorithms apply a rather naive strategy to prevent race conditions: they rely on *mutual exclusion* (mutex) of access to shared resources. Mutual exclusion can be implemented by first defining a shared "lock"-variable or *semaphore*. When a thread wants to modify any shared data, it will first call a function implementing a mutex that will use the semaphore to indicate that access to the data is locked. When this function returns it can begin executing all code that require these shared resources. This code is also known as the critical section. Subsequent threads that try to call this mutex function will now be stalled until the first thread calls a second routine indicating it no longer requires exclusive access to the shared resources, after which the next thread to obtain the lock can execute its critical section. Although this is an effective technique to prevent race conditions and to protect shared data, it does not scale well on a parallel platform because a lot of time could be wasted by threads that are stalled until another thread finishes.

Lock-free algorithms follow a different strategy. Instead of ensuring a critical section is only executed by one thread at any given time, lock-free algorithms construct the critical section in such a way that it is safe to execute in parallel. The common way to design a lock-free algorithm is by making use of the atomic Compare-And-Swap (CAS) operation[22] available on many modern processors. As the name implies, the operation first compares the value of a given memory location with one of the parameters. Iff these values are equal, the new value is written to that address. The old value is returned, which will give the program the opportunity to verify whether the CAS operation had succeeded or not. The CAS operation is atomic, meaning it cannot be interrupted.

Normally this CAS operation is used to change the value of a shared variable. The idea behind lock-free algorithms is to consider the CAS operation as a checkpoint, where the purpose of this CAS operation is to commit all work up to this point. Every time a CAS operation is executed by one or more work-items, there is at least one thread that makes progress because its CAS operation succeeds. All the other threads might need to re-do some work when the CAS operation fails, but never more than the work between the previous checkpoint until this one. If a program using this technique always adheres to an invariant that holds after every CAS operation, it can be shown that the parallel algorithm manipulating global data indeed behaves correctly.

Lock-free algorithms have the advantage of not mutually excluding threads from access to a resource, thus not forcing threads to operate one by one. Unfortunately, these algorithms are a lot more difficult to construct due to the complexity of its invariant. For a lot of data structures no complete lock-free algorithm exists currently. Furthermore, although lock-free algorithms are presented as a promising alternative to locking algorithms and yield good performance on MIMD machines, lock-free algorithms could fall short on SIMT machines. Because on a SIMT machine a core is executing the same instruction at the same time as other cores in the its cluster, the failure rate of a CAS instruction is likely to be significantly higher than on a system with autonomous cores.

Many lock-free algorithms suffer from a problem that in literature is called the "ABA"-problem. To understand this problem, consider a queue with nodes $b1 \rightarrow b2 \rightarrow b3$. In this example thread 1 tries to dequeue element *b1* from the queue. Before committing the change to the global data structure, it gets pre-empted by thread 2. Figure 2.1 shows a possible execution flow.

Thread 1	Thread 2
block = *head (b1) next = block→next (b2)	
	block = *head (b1) next = block→next (b2) CAS(head, block, next)
queue: $b2 \rightarrow b3$, head: b2	
	block = *head (b2) next = block→next (b3) CAS(head, block, next)
queue: $b3$, head: b3	
	enqueue(b1)
queue: $b3 \rightarrow b1$, head: b3	
	block = *head (b3) next = block→next (b1) CAS(head, block, next)
queue: $b1$, head: b1	
	enqueue(b3)
queue: $b1 \rightarrow b3$, head: b1	
CAS(head, block, next)	
queue: $b3$, head: b2	

Figure 2.1: Example ABA problem

In this example thread 1 was interrupted after reading any relevant values, but before committing its changes. During this interruption thread 2 dequeued and re-enqueued b1. This particular sequence of operations led b1 to no longer point at b2, but at b3. When thread 1 continues operation, it finds b1 at the head of the queue, and therefore assumes nothing has changed. It will successfully execute the CAS operation, and as a result *head* now points to b2. However, b2 is no longer on the queue.

The most practical work-around for this problem, used by M. Maged for implementing a queue data structure[32], is by the use of tag counters in the field. A pointer is defined as an $\{address, tag\}$ tuple. By incrementing this tag on every operation, the b1 pointer that appears at the head of the stack after thread 2 is finished most likely differs from the value in the *block* variable of thread 1. The more bits are available for the tag, the less likely it is to wrap around and cause an undetectable ABA problem. With such a queue implementation, the CAS operation executed by thread 1 would have failed, preserving a correct stack state and forcing thread 1 to try again.

2.2 Memory allocators

One of the features covered by the POSIX C standard is the heap allocator. This feature enables programmers to allocate memory dynamically instead of statically by calling malloc(). The difference between static and dynamic allocation is illustrated in the following example C code:

```
void function() {
    int staticArray[10];
    int *dynamicArray;

    dynamicArray = malloc(sizeof(int) * 10);
}
```

Functionally there is no difference between staticArray and dynamicArray; at the end of the function call both variables point to an array of 10 integers. But while the size of staticArray is decided at compile time, dynamicArray points to a block of memory allocated at run-time.

```
void function(unsigned int entries) {
    /* This results in a compile error: */
    int staticArray[entries];
    /* While this is valid: */
    int *dynamicArray;

    dynamicArray = malloc(sizeof(int) * entries);
}
```

This second example shows the benefit of a memory allocator for programmers. Because the size of a memory block can be determined at run-time when using *malloc()*, an array could for instance consist of an arbitrary number of integers read from a file or socket. Memory allocated with *malloc()* is not limited to any content type. After usage, the memory can be returned for use by another program by calling the *free()* routine.

The POSIX standard defines that the heap allocator must work as shown above, but places no restrictions on the implementation of the heap allocators routines. Some big software projects replace the allocator they use in their program from the one available in the operating systems libraries to an implementation that better suits their demands[9].

Terminology among these different implementation appears to be somewhat consistent. A program allocates a *block* of memory by calling malloc(). Depending on the size of this block and the implementation, this block could be allocated directly from the operating system, or allocated from a *superblock*. A superblock is simply a chunk of memory that contains several blocks, allocated from the operating system in its entire and managed by the heap allocator. When a heap allocator decides it no longer requires a (super-)block of memory, it *returns* this block to the OS.

From the allocators point of view, obtaining a superblock and distributing blocks from it is more efficient than obtaining each block separately from the operating system (*OS*), because it reduces the number of costly software-interrupts of the OS. Furthermore, the OS can often only provide full *pages* of memory, which on an x86 system are typically 4KB each. Because a lot of user applications allocate blocks of memory smaller than 4KB, this superblock system eliminates the wasting of a lot of memory. This wasted memory is one form of *fragmentation*, which inherits its name from the availability of an unused fragment of memory between two allocated blocks.

Superblocks can be seen as a form of caching: the heap allocator provisions memory to have it ready for use when the program demands it. Some memory allocators also cache empty superblocks instead of returning them immediately to save more *round-trips* to the OS. Caching an empty superblock is usually done by adding it to a *free-list*. In a multi-tasking environment any memory returned to the OS can be re-used by another program. It is thus important to find a good balance between the performance benefit gained from caching superblocks, and the reliability of the other programs running on the machine gained from returning superblocks.

The current values of the free-list and any other variables that the heap might use for administrative purposes is called the *state* of the heap.

Fragmentation One of the qualities of a good memory allocator is its ability to fight wastage of memory caused by fragmentation.[40] Two forms of fragmentation are identified: *external* and *internal* fragmentation.

External fragmentation occurs when two blocks of allocated memory have an unallocated piece of memory in between. This unallocated piece of memory might be available for re-use, but its size could be a limiting factor in the usefulness of this block. External fragmentation usually is caused by freeing of blocks.

Consider the case where three blocks of 20 bytes were allocated. If the middle block is deallocated, a gap of 20 bytes arises. A new allocation request of 18 bytes might be fulfilled from

this gap, reducing the size of the gap to 2 bytes. Because allocation of two bytes hardly occurs, two bytes of memory are wasted on external fragmentation.

Internal fragmentation is the name of any excess bytes caused by rounding up the size of a block. A heap allocator might decide to round up the block size for any request if this leads to more regular block sizes. This effectively fights external fragmentation at the possible cost of this internal fragmentation. Internal fragmentation is usually seen with memory allocators that use a superblock system where each superblock serves the requests of a specific size class.

One of the size-classes in a superblock-based memory allocator could be "12 to 16 bytes", where each block in this superblock is of 16 bytes in size. If a program requests precisely 16 bytes, no internal fragmentation will occur. However, if a program requests 12 bytes, an internal fragmentation of 4 bytes is observable between the end of the current block and the next.

Parallel systems In parallel systems, the heap state is a combination of shared variables. Without taking proper precautions, this heap state is prone to race conditions. In parallel programs that rely heavily on memory allocations, such as database systems or web servers[7], memory allocators can easily become the bottleneck. The performance of a heap allocator is greatly influenced by the process stalls resulting from the use of mutexes.

The challenge of creating a fast memory allocator lies in preventing threads from stalling. This involves finding a good alternative for a mutex. Alternatives found in literature include reducing the use of shared resources[7] and designing a lock-free allocation algorithm[16, 41].

2.3 GPU

The graphics card has traditionally been a device that serves two purposes: producing images and bringing these images to the screen. The latter task is performed by a lot of infrastructure inside the graphics card to drive the output ports and to read out memory. The first task has throughout the years grown to be supported by a massively parallel platform. At the heart of a GPU nowadays lie many computational cores, ranging from 48 units on a low-power laptop GPU to several thousand on the more expensive high-end GPUs.

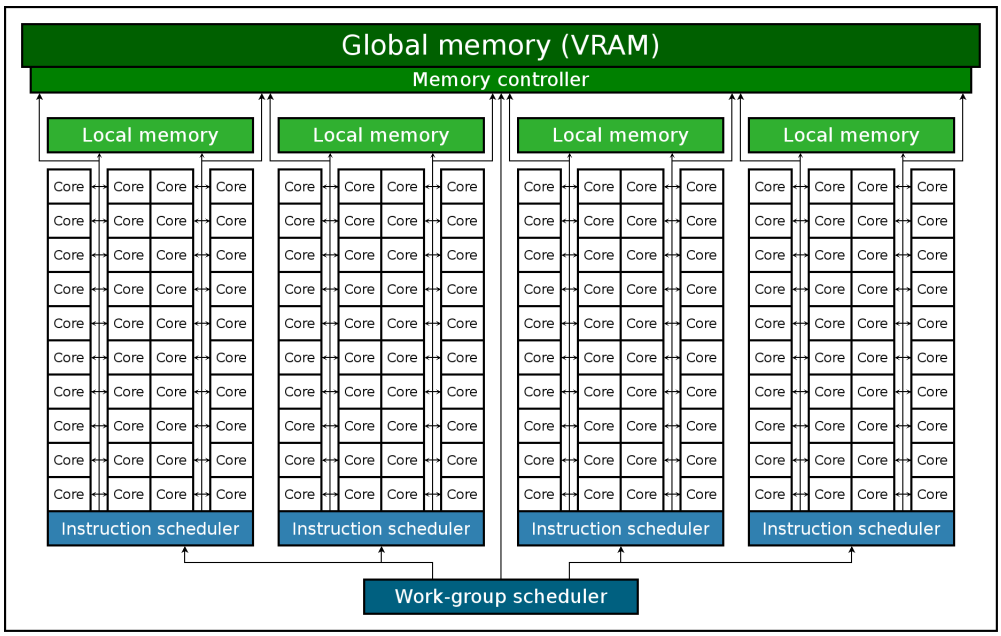

Figure 2.2: Modern GPU architecture - simplified

Graphics programmers can use these cores by writing "shaders"[39], small programs that contain the algorithm for a single thread. These shaders are used to transform vectors into pixels, and then to manipulate each pixel for instance by applying shadow or lighting. Because there are many vectors and pixels to work with, there is a lot of potential concurrency. The more cores a GPU has, the faster it can finish these computations. This speed translates to the user in higher quality details, a higher resolution and a higher frame rate.

Modern GPUs are organised like depicted in figure 2.2. Cores are generally clustered in groups, sharing some resources like an instruction scheduler and some (shared, on-chip) local memory. Sharing the instruction scheduler means each core in a cluster will execute the same instruction.

A lot of the complexity of a GPU lies in the memory subsystem. By using reordering and grouping techniques as well as effective local caches, GPUs are able to achieve sufficient bandwidth for processing all the data required to produce a high-resolution picture.

The memory model of a GPU consists of at least three layers: global memory (VRAM), per-cluster memory and local registers[34]. Here global memory is the slowest to access with an added latency of 400 to 800 cycles, but the largest in size. Local memory is reported to be up to 100 times faster than global memory, and local registers inside each core have no extra latency at all if read-after-write hazards are avoided. In addition to this memory hierarchy many GPUs feature L1 and L2 caches to speed up memory reads from frequently accessed regions[44]. Modern GPUs and computers also support GART[14], a method to map system memory into the virtual memory space of the GPU. From the GPUs point of view this method is even more costly to access than VRAM as all access has to pass through the systems memory bus, but it eliminates the need to do an initial upload or finishing download of data to the GPUs memory.

To get a GPU to execute a shader, the host system first uploads or maps the code and the required datasets to the GPUs memory. It then issues a launch of this program, labelled with a specific identifier. When computation is done, the GPU interrupts the host system to notify execution of the program has ended, after which the host system can freely download and/or further process this data.

A GPU is designed to be a slave of the host system, not an autonomous device. It can use interrupts its memory-mapped (MMIO) interface to inform the host system of its status, but for performance reasons it is undesirable that a GPU waits for the host system. The I/O capabilities of a GPU are also not as extensive as those of a CPU; it cannot access other devices like hard disks or network devices. All the data a GPU has access to must be uploaded or mapped into the GPUs address space by the host system.

This architecture implies several limitations when compared to a CPU. Programmers must take special care for these limitations when designing their system. The most important limitations in the light of memory allocation are:

Shared program counter One of the pieces of hardware that is shared between multiple cores in many modern GPUs is the program counter. As a result all cores in a cluster will always execute the same instruction, typically on different data. This is referred to as SIMD or SIMT execution[33, 44].

For conditional code, this means that the cores in a cluster cannot simply start executing different instructions. Instead, all the threads that do not meet the condition must be disabled (masked out) to prevent execution of this code[34]. This mechanism makes divergent branches expensive on a GPU. In the case where there are two code paths to be taken and execution on different cores diverge, the execution time of this block of code for the group of cores will be the sum of the execution times for both paths. With a large number of different code paths, code wil run as if it is serialised. With clusters of cores typically containing 32 cores sharing one program counter, this means the execution takes up to 32 times as long as when code paths never diverge.

GPU-to-Host communication The available global memory is managed by the device driver on the host system. If a program running on the GPU wishes to allocate a block of memory, theoretically this means that the GPUs program flow must be interrupted to notify the host

operating system that memory is required. The device driver can then allocate this memory and return a pointer to the allocated block, after which execution can be continued. There are two reasons why this is undesirable in practice: (1) interrupting the operating system is a costly operation because the GPU will be idle waiting for the host system to return its memory and (2) the required communication path to interrupt the host system from with a program running on the GPU is currently not implemented in modern graphics cards.

Virtual memory space On a CPU there are two distinct memory spaces. First there is the *physical* address space, the direct mapping of a memory address to either memory or the MMIO space of a device in the system. On top of this, a *virtual* memory (VM) address space is placed that maps to physical addresses in any way the operating system sees fit. Each process on a system has its own VM space. By using this mechanism, the operating system can prevent a process from accessing memory that belongs to another process. Besides, an operating system can pick several blocks of memory spread physically, and map them contiguously in the virtual address space such that a process can allocate large chunks of memory even when physical memory is highly fragmented.

A GPU has its own VM layout which differs from the VM layout of a process on the host system. Blocks in the GPUs VM space can be mapped to either the GPUs own VRAM, or to the hosts memory using GART. On some graphics cards the VM space even differs between the various processes or contexts running on the GPU, much like on a CPU.

The difference in VM layout between GPU and CPU comes with additional challenges. Because the layout is different, pointers in one address space does not necessarily point to the same object in the other. Because linked lists and trees work with pointers to other nodes, they cannot be interpreted after transferring to a different device. Currently no driver gives the host the freedom of parsing the virtual memory pagetables of the GPU, so the context of this structure gets lost on transfer.

2.4 OpenCL

With the introduction of platforms like Cuda[36] and OpenCL[27], the GPUs massive-parallel computational power became available for more general purposes. Both platforms describe a host-side API that lets a program control a special compute environment, and a programming language that lets a programmer write code to execute on the target device.

The difference between Cuda and OpenCL is both of technical and political nature. Cuda is an environment developed by NVIDIA to fully utilise their GPU hardware. OpenCL on the other hand is an open standard driven by most major players in the industry, including AMD, NVIDIA, Intel, ARM, Apple and many others. One of the key properties of OpenCL is portability: OpenCL code can be executed on a variety of hardware, such as CPUs, GPUs and FPGAs, from many vendors. Although Cuda feature-wise has an advantage over OpenCL[36, 27], studies show that programmers are able to obtain a similar performance with both[18] when targetting the same device.

With its specifications released in 2008, OpenCL is a relatively young standard. Its programming model is designed to closely match the GPU hardware, without compromising in portability to other platforms like the CPU. The most recent version at the time of writing is OpenCL 1.2. Some of its features are missing in the current implementation of NVIDIA, which only promotes OpenCL 1.1 compliance. Most of the concepts in OpenCL should sound familiar to OpenGL programmers, although terminology matches more with that of traditional programmers.

The memory model of OpenCL describes the use of global memory, local memory and local variables that map to local registers. This corresponds to the general architecture of a GPU as shown in figure 2.2. To allow performance optimisation by the device it also supports buffers like read-only constant buffers and two- or three-dimensional image objects, both of which reside in global memory.

On the host side OpenCL offers an extensive API to set up an OpenCL environment for a program. A typical OpenCL work-flow looks as follows:

1. Select a *platform* (NVIDIA Cuda, AMD APP, Intel SDK...)
2. Choose a *device* for this platform (GPU, CPU)
3. Create a *context* on this device
4. Create a *command queue* for this context
5. Create and compile the OpenCL *program* from sources
6. Create a *kernel* object from this program, specifying the desired kernel
7. Create input and output *buffer*s for the *kernel*
8. Schedule the upload of data to these buffers in the *command queue*
9. Set the *kernel parameters* in the kernel object and schedule *kernel* execution in the *command queue*
10. Wait for the *command queue* to run empty, indicating execution has finished
11. Read back the results

Each platform offers support for one or more devices, but devices are not limited to a single platform. For instance an Intel CPU can be user by both the Intel and AMD platforms, each using their own compiler back-end. After choosing a device, the host program creates a context on it. This context is used to tell apart the different programs utilising the same device.

OpenCL programs are typically written in a special subset of C, called OpenCL C. Such a program contains all the work that one thread, or work-item, must complete. This programming model, called SIMT, closely resembles shaders in graphics processing. A program contains one or more entry points that the host can use to start execution. These entry points are special functions called kernels. By running the same OpenCL kernel on different cores with different input data, parallel execution is achieved with this sequential code. Programmers are responsible to handle or eliminate data dependencies between work-items.

In order to launch a kernel, the OpenCL API demands a thread configuration. This configuration specifies how many threads (work-items) are executed and how they are grouped in work-groups. To allow efficient optimisation by the GPU, this configuration is given in one or several dimensions and should map to the data set processed.

Because OpenCL is a fairly new standard, it comes with quite a few restrictions. Some of these limitations are imposed by hardware while others are simply a matter of specification. The rest of this subsection will cover some of the restrictions encountered in OpenCL during the creation and testing of prototypes on NVIDIA hardware.

Locking In a lot of shared data structures, the state of the structure is protected by means of a mutex lock as described in section 2.1. The OpenCL specification does not include such a lock, but by using the available atomic operations in OpenCL it seems possible to create a fairly simple spin-lock that works on a regular MIMD machine. However, for a SIMT device like a GPU writing a spin-lock prove to be not nearly as trivial. This is caused by the fact that a GPU will not have a program counter or instruction dispatcher per core, but rather share the same between many cores. To illustrate the problem, consider the following simple example for implementing a spin-lock:

```
/* Global vars */
global volatile int lock; /* Initialised to 0 */

function doCritical() {
  int oldvalue;
  do {
    oldvalue = atom_CAS(&lock, 0, 1);
  } while(oldvalue != 0);
  /* Critical section here */
  atom_set(&lock, 0);
}
```

On a MIMD machine supporting the atomic CAS operation, this function will attempt to swap the lock from 0 (unlocked) to 1 (locked). If a particular thread succeeds it continues to execute the critical section, after which it releases the lock again. However, on a GPU the critical section will never be executed because the one work-item in a work-group that obtains the lock will be masked out from further execution. The scheduler will instead make the remaining cores loop over the while-loop until an active core sets the lock value to 0, an event that will never occur.

Attempting to "trick" the scheduler into masking out all cores but the one that obtains the lock prove not to work. Likely the compiler tries to optimise the code instead and treats the second piece of code like the first one.

```
/* Global vars */
global volatile int lock; /* Initialised to 0 */

function doCritical() {
  int oldvalue;
  while(true) {
    if(atom_CAS(&lock, 0, 1) == 0) {
      /* Critical section here */
      atom_set(&lock, 0);
      break;
    }
  }
}
```

Because shared program counters are a fundamental property of many GPUs[44, 24], more complex code constructions based on this atomic principle are unlikely to lead to a reliable and portable lock mechanism.

Memory ordering GPUs make heavy use of memory reordering by design, because it allows for a much greater utilisation of the available memory bandwidth. Limiting reordering by explicitly enforcing a memory order in source code likely degrades performance of the GPU in favour of correctness. Unfortunately, memory ordering guarantees are sometimes required to be able to construct reliable algorithms where threads co-operate on the same shared data structure.

The OpenCL standard states the following about memory ordering:

> OpenCL uses a relaxed consistency memory model; i.e. the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.[27]

This relaxed consistency memory model implies that there are no hard guarantees about memory ordering. Without these guarantees, it becomes quite difficult to work on true global shared

memory structured like lists and trees. Lock-free parallel algorithms for instance rely on certain check-points where a set of changes is committed and verified using the atomic CAS operation. As the order of execution for these atomic operations is not predictable, the complete system could read a state which is impossible if the code was executed sequentially. Accounting for all these possible states is not desirable and often not even possible in a lock-free algorithm.

In practice it seems on NVIDIA hardware and CPUs that support AMD APP, the order of memory operations can be controlled by using memory fences. Although the specification is unclear about whether these operations are safe to use to synchronise memory operations between different work-items, tests on this hardware with a global queue data structure seem to indicate it is. On recent AMD HD graphics cards, memory fences do not influence the memory ordering like on NVIDIA or CPUs supported by AMD APP. The behaviour of algorithms that require a specific memory ordering is therefore unspecified on these GPUs.

Global synchronisation Although synchronising is an expensive operation because all threads have to wait for each other, it is a necessary evil for reduction operations. OpenCL offers barriers that allow threads in a work-group to synchronise at a given point in source code. Unfortunately, no such barrier exists for global synchronisation of all threads.

As an experiment we have written a global barrier function in OpenCL C. A similar experiment has been conducted in Cuda by Shucai Xiao[45]. Consider the following snippet of pseudo-code, which is safe when re-entry is prevented:

```
global_barrier(volatile unsigned int __global *b)
{
    if(pid_local() == 0) atom_inc(b);

    /* Synchronise locally; a lot cheaper this way */
    barrier(CLK_LOCAL_MEM_FENCE);

    /* Wait for all the other workgroups */
    while(*b < workgroups() && *b > 0);

    /* Reset b, _one_ thread only */
    if(pid_global() == 0) atom_and(b, 0);
}
```

This code uses the regular barrier operation to synchronise the different threads in a work-group. It then increments a counter and waits until the counter is equal to the number of work-groups. At that point all threads should be synchronised, and all that is left is to reset the counter once.

Our experiments on NVIDIA hardware show that global synchronisation works for a small number of threads with a very specific thread configuration. On an NVIDIA GeForce GT640 the work-item scheduler schedules around 3000 threads at the time. However, there are many variables that influence this number of work-items, making it impossible to predict when this code will or will not work in a portable way. Similar results were obtained in [45].

The reason why this method does not work with too many work items is because scheduling techniques are used to keep register pressure low. Instead of executing all work-groups in round-robin, NVIDIA hardware schedules a subset of the work-groups. Once a work-group is finished it schedules a new one for execution. This technique eliminates the need to store register values on a context switch, and at the same time makes more local registers available to a work-item reducing register spills. The downside of this technique is that beyond a certain amount of work-items, at no point in time all the work-groups will gather inside the barrier function. As a result a synchronisation function like above leaves the active work-groups busy waiting forever.

The only way to enforce global synchronisation in OpenCL is by splitting up the kernel in two halves. The top half can then be finished before the host system launches the bottom half. When

executing loops or reductions however this implies a lot of overhead and could greatly increase the complexity of the code.

64-bit atomics Depending on the platform a memory allocator has to deal with either 32-bit or 64-bit pointers. In lock-free implementations of data-structures like singly linked-lists or even hash-tables, pointers will need to be updated atomically to verify expected execution and to be able to detect situations where another thread updated the pointer before. Without these verifications there is no way for a lock-free algorithm to ensure a consistent state.

The OpenCL 1.2 standard includes 32-bit atomic operations, but does not define 64-bit counterparts. Atomic operations on 64-bit atomics are defined in separate extensions. Unfortunately not all platforms implement these extensions, most notably Intel SDK for OpenCL Applications 2013. A 64-bit atomic operation is mandatory for any memory allocator that works on platforms with 64-bit memory addresses. For GPGPU platforms this is not necessarily a restriction, as both AMD and NVIDIA GPUs expose a 32-bit virtual memory layout to the OpenCL kernel. For execution of OpenCL kernels on the CPU on the other hand this means that the entire application needs to be compiled with 32-bit support unless 64-bit atomics are supported by the compiler. Intels SDK also does not support 32-bit applications, making it impossible to create a memory allocator that works on this particular platform.

2.5 Related work: Heap allocator algorithms

As memory allocation is an important and popular feature of many programming languages, a lot of research has been done to find an efficient allocation algorithm. Many of such algorithms have been developed to offer fast memory allocation while wasting as little memory as possible. In the literature two heap management algorithms are often found: Doug Lea's DLmalloc[29] as used in the GNU libc implementation, and the GPL-licensed Hoard algorithm[7]. We consider both algorithms a good candidate to base a memory allocator for OpenCL on.

DLMalloc Lea's allocation scheme is a combination of two different algorithms[29]. It uses one heap for all threads, and free blocks are put together in size-binned linked lists. For small and medium sized allocations a best-fit algorithm is used. Large objects are obtained directly from the operating system.

The best-fit algorithm works as follows: when available a block is returned from the correct size bin. If no such block is available, a bigger block is taken from a different size bin and split up into two smaller blocks, after which one block is returned and the other one attached to the right size bin. When a program frees a block it is coalesced with adjacent free blocks, forming one large block. Blocks start at 16 bytes, the minimum required to link them together when free, and increase in 8-byte steps.

By using a large amount of size bins there is only a very limited amount of space wasted due to returning blocks that are too large (internal fragmentation). Excessive 8-byte blocks can always be split off as a separate block and added to the free-list, further reducing internal fragmentation. However, DLMalloc is prone to "external fragmentation", which occurs when blocks of memory in the free-list are too small for the program to use, yet cannot coalesce into larger blocks because their adjacent blocks are both taken.

Hoard Emery Berger et al. propose the Hoard allocation algorithm [7]. At the heart Hoard manages superblocks containing equally sized memory blocks. What makes Hoard unique is its improved performance on parallel systems. This is achieved by besides managing a global heap, each process also has its own local heap. By limiting this local heap to one thread, there will be no possible race conditions. This means that for any operation on the local heap there is no need for mutexes or similar protection measures, so threads no longer always have to stall when another thread is busy with its allocation- or free-routine. Only then when the local heap runs empty will the global (thread-safe) heap be used.

Dave Dice et al. proposed a more efficient mostly lock-free heap allocator based on Hoard[16] for Solaris systems. This heap allocator is mostly lock-free by relying on Solaris' process scheduler to warn a process when its allocation was pre-empted, forcing a restart of the routine. Michael Maged proposed a portable and fully lock-free allocator based on Hoard that makes use of the Compare-And-Swap (CAS) instruction of many modern processor architectures [31]. Benchmarks show that this allocator scales nearly perfectly up to 16 processors, without introducing any measurable latency overhead.

Each superblock falls into a certain "size-class", where each class covers a range of requested memory block sizes. Memory blocks larger than the upper bound of the largest size-class will be allocated directly from the operating system.

By allocating and grouping blocks of memory of the same size, external fragmentation hardly exist. On the downside the amount of required memory in a request must always be rounded up to the size class border, causing internal fragmentation. Hoard chose the size classes in such a way that this internal fragmentation is at most 20% of the available memory space. In addition, because each size class gets allocated several 4K memory pages from the operating system, a lot more space is wasted when the required heap space is small. In a multi-tasking environment this space could have been used by other applications.

Although this strategy proves to be very efficient on MIMD machines like multi-core CPUs, there are two reasons why this strategy is not efficient when applied on a GPU. First of all, to fully utilise the available computational power of a GPU it is advised to spawn more threads than the number of available cores[34]. If 3000 threads are spawned, and each of these threads maintains a heap with one 4KB superblock, the entire program already uses a heap of 12MB. Because the memory requirement of a single work-items is likely small, most of these superblocks will be underutilised. This results in a lot of wasted memory.

The second reason why per-work-item heaps are a bad idea on a GPU is related to the alignment of superblocks. Figure 2.3 illustrated a memory alignment likely to occur in Hoard.

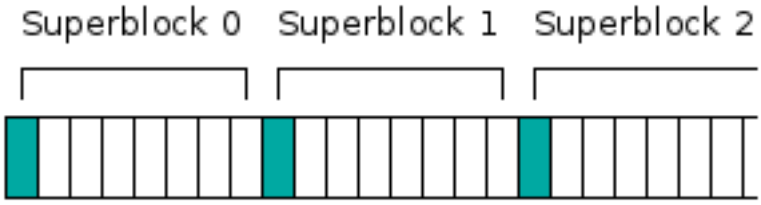

Figure 2.3: Possible memory alignment with Hoard

Consider the case where all threads allocate a block of memory at the same time. With Hoard, each thread will obtain its own superblock and allocate a block from it. Allocation from this superblock will follow a fixed pattern, usually starting at the first block inside this superblock. The result is a strided memory access, where the stride is the size of the superblock. Hardware cannot combine this access into larger requests because blocks are not adjacent, resulting in a lot of small memory requests each with a relatively high latency. This access pattern prevents the GPU from fully utilising the available memory bandwidth when these objects are accessed in parallel, greatly degrading performance. To make matters worse, superblocks are often of size 2^n. In these situations the L2 cache is not very efficient either unless alignment is forcibly broken. It can be argued that memory read performance of memory blocks distributed by an implementation of Hoard can not be portable between different devices, because caching becomes ineffective.

XMalloc Xiauhuang Huang et al. proposed the XMalloc allocator[23] for Cuda, an allocator designed to work around issues with lock-free algorithms on SIMD and SIMT machines. XMalloc structures the available memory in a three-level hierarchy: superblocks contain basic blocks, and basic blocks consist of several memory blocks. Free superblocks are connected to a free list, which is a fixed size fifo. Another fixed size fifo exists for free basic blocks. Unlike Hoard, XMalloc only

works with a single global heap.

Often in a kernel all work-items in a work-group call the malloc()-routine at the same time. XMalloc optimises these requests by, instead of having each work-item allocate a chunk of memory, gathering the memory requirements of all work-items by means of a prefix-sum reduction. XMalloc first gathers the total memory requirement for all work-items by applying the up-sweep phase of the prefix-sum algorithm[10]. It then allocates the total amount of memory required to serve all work-items. Finally, this memory is distributed by the down-sweep phase of the prefix-sum addition.

This prefix-sum reduction has a time complexity of $O(logn)$. Benchmarks show a great improvement in latency of this approach compared to the linear CudaMalloc[36] and excellent scaling with the number of cores in the SIMT machine.

Chapter 3

Memory allocation patterns

To reach the goal of designing a memory allocator suitable for use in GPGPU-enabled applications, the following questions must be answered:

- What are "GPGPU-enabled applications"?
- What are the demands on a memory allocator for these applications?
- How can we meet these demands on a GPGPU architecture?

We believe the best way to answer these questions is by conducting a use-case study. By systematically quantifying the behaviour of available programs regarding the lifespan of allocated memory (allocate, read/write, free), many conclusions can be drawn about the desired use of a memory allocator in OpenCL.

To ensure that the investigated set of programs is diverse, we attempted to find implementations of different algorithms. Krste Asanovic et al. proposed a series of twelve basic patterns that cover most of the available parallel problems seen today[3]. In the rest of this document these patterns will be referred to as algorithm classes.

In our survey we do not limit ourselves to GPGPU-enabled programs. OpenCL programers had no access to a heap allocator while Cuda programmers are discouraged the use of their allocator. Although programmers could have worked around this absence, code using such work-arounds are not always representative for the code its programmer envisioned. Besides investigating OpenCL and Cuda implementations we will thus also consider OpenMP implementations of common massive-parallel problems.

3.1 Research setup

For the design of a memory allocator two factors are of particular interest. First of all good insight is needed in the allocation process. Is the provided malloc()-routine called often or even per-iteration, or rather just once or twice at the beginning of a program? And related to this, is the memory free'd all at once, or in a different pattern? How large are the actual allocated blocks? Are these allocated blocks (arrays of) equally-sized objects?

The second factor of interest is the use of these allocated blocks. When allocated memory is re-used a lot, it is relevant to see what pattern can be observed in the usage of these allocated blocks. Having simultaneously accessed data close together is one of the optimisations recommended to improve performance[34] of OpenCL kernel execution. Xmalloc uses the prefix-sum algorithm for distributing the allocated blocks[23], but this might not be the most efficient distribution when considering the limited memory bandwidth on a GPU. How does a program access the allocated memory? Does it only need its own data? Is it even possible to improve on performance by using a different distribution method?

Based on these questions, we propose to extract the following metrics from each program to profile dynamic memory access behaviour:

- Number of calls to malloc() in terms of p (processors) and n (data size)
- Number of calls to free()
- Size of each allocated object
- Number of allocated objects
- Access pattern[26]
  - Linear
  - Reverse linear
  - Shifted
  - Overlapping
  - Non-unit stride
  - Random
- Access globally or locally
- Read/write frequency

The required data will be extracted from a couple of sources. Ideally this study consists of analysing the source code of a wide variety of parallel programs, each designed for a platform that has a memory allocator available. If source code for a given algorithmic class is unavailable, we either use details from implementation papers or an analysis of the algorithm to show the usability of a malloc() routine.

One of the sources for parallel implementations is the Rodinia benchmark suite[13]. This suite contains a variety of benchmarks covering five out of the twelve algorithmic classes, each implemented in OpenMP, Cuda and OpenCL. We have picked three of the programs, as a quick look at the source code revealed that the rest of the programs were designed for a Cuda environment without memory allocation. This implies that the applications are biased towards a malloc-free environment, and thus will not generate useful results for this study. The selected programs are ones based on ready-available GPL source code, rather than designed specifically for Rodinia.

Parboil is another benchmark suite[2] containing a wide variety of parallel programs. Although this suite does not contain any parallel implementations for the CPU, it does contain several Cuda kernels designed for the Cuda 3.2 toolkit, thus supporting malloc in kernels. Unfortunately this malloc routine is never used in any of the kernels. The fact that this routine is not used can be an indication that programmers do not desire a dynamic memory allocator for use in their kernels. Nonetheless we will investigate several Cuda implementations to find variables that are in theory suitable for allocation using malloc(). We acknowledge that this data is a source of discussion and will show that this discussion is actually a good reason not to disregard this data.

Several other sources are used to find good parallel programs for investigation. A complete list of the chosen programs and their sources can be found in table 3.1. For all of these programs, all relevant allocated variables are identified and their usage is written down in terms of the earlier mentioned metrics. Data in an OpenCL program that needs to be shared between the host and the accelerator can not be allocated on the accelerating device due to a difference in VM spaces, as explained in section **??**. The memory allocation behaviour for these datasets is therefore not considered in the results of this study. The remaining variables are documented in table 3.2.

For the Finite State Machine algorithmic class, we extracted relevant data from a case study written to describe the research towards accelerating such an algorithm by means of exploiting parallelism[42]. Although this papers does not describe how each variable in the algorithm is

implemented, it does explain in detail the challenges encountered while implementing their proposed solution. From the paper it can be derived what data structures are used and how they are implemented in their prototype.

Finally, for the Dynamic Programming algorithmic class it is sufficient to reason from the theoretical side how it should be implemented in a parallel environment. Algorithms that fall into this class are well defined for sequential operation, but as we will see parallel implementations often face challenges related to scheduling.

Algorithm Class	Program	Source		Library
Finite State Machine	Level-7 filtering	Case	Hellas University[42]	-
Combinatorial	?	?	?	
Graph Traversal	Graph analysis	Code	TU Delft	OpenCL
Structured Grid	Heart Wall	Code	Rodinia	OpenMP
Dense Linear Algebra	K-Means	Code	Rodinia	OpenMP
Sparse Matrix	SPMV	Code	Parboil	Cuda
Spectral (FFT)	FFT	Code	Parboil	Cuda
Dynamic Programming	Dijkstra	Theory	-	-
N-Body/Particle Methods	Barnes-Hut	Code	Texas State University[1]	OpenCL
MapReduce	?	?	?	
Backtracking	?	?	?	
Unstructured Grid	Back Propagation	Code	Rodinia	OpenMP

Table 3.1: Selected programs for use-case study

[1] http://www.gpucomputing.net/?q=node/1314

3.2 Results

Variable	Malloc()	Free()	Obj. size	Obj. count	Access[2]	Access Pattern	R/W freq
3.2.1: Finite state machine							
work_queue	p*i	p*i	state	1	r: global w: local	-	O(i)
3.2.2: Graph analysis							
nodetree	p*i	p	node	p/i	global	random	$O(nlogn)$
3.2.3: Heart Wall							
private[*p*].*d_in*2	p	p	float	const[3]	local	r: linear, non-unit stride w: linear	$O(i^2)$
private[*p*].*d_in*2_*sqr*	p	p	float	const[3]	local	linear	$O(i)$
private[*p*].*d_in_mod*	p	p	float	const[3]	local	r: linear, non-unit stride w: linear	$O(i^2)$
private[*p*].*d_in_sqr*	p	p	float	const[3]	local	linear	$O(i)$
private[*p*].*d_conv*	p	p	float	const	local	linear	O(i)
private[*p*].*d_in*2_*pad*	p	p	float	const[3]	local	linear, non-unit stride	O(i)
private[*p*].*d_in*2_*sub*	p	p	float	const[3]	local	linear, non-unit stride	O(i)
private[*p*].*d_in*2_*sub*2_*sqr*	p	p	float	const[3]	local	linear	O(i)
private[*p*].*d_tMask*	p	p	float	const	local	linear, random	O(i)
private[*p*].*d_mask_conv*	p	p	float	const	local	linear	O(i)
3.2.4: K-Means							
membership	p	p	int	1	local	-	O(n)
new_centers	1	1	float	$k * dim$	global	linear	O(i)
partial_new_centers	p	p	float	k	global	r: non-unit stride w: linear	O()
partial_new_centers_len	p	p	int	k	global	r: non-unit stride w: linear	O()
3.2.5: Sparse Matrix Vector							
fictive intermediate	1	1	float	$dim(X * Y)$	global	random	?
3.2.6: Fast-Fourier Transform							
v	p	p	float	const	local	random	O(i)
d_work	p	p	float	const	local	random	O(i)
3.2.8: Barnes Hut							
octree	p*i*j	p*j	node	$n(*j)$	global	random (local)	$O(nlogn)$
3.2.9: Back propagation - no applicable variables							

Table 3.2: Identified variables in parallel algorithms and implementations

3.2.1 Finite state machines

State machines are a basic mechanism for processing a string of input characters. By evaluating the next input character and the current state, a new state can be determined. In particular, any regular expression can be represented by a state machine.

There exist very few parallel programs that process a state machine. One experiment with regular expressions on GPUs has been done at the university of Hellas[42]. They have created

[2]Local: only by the thread allocated; Global: data used by any thread
[3]Could be implemented as rand()*n for the border of a video

an implementation of "level-7" filtering or "deep-packet inspection" in networks. Each network package needs to be tested against a large set of regular expressions to determine the content type of the package. Each regular expression gets converted to a state transition table and will be placed in (read-only) global memory. On the GPU every data packet can then be processed in parallel, trying to match the data against all available regular expressions.

Lacking source code, we will investigate finite state machines on a theoretical level while considering the decisions explained in the research on "level-7" filtering[42]. In FSM algorithms, the number of iterations i is equal to the length of the input word. The number of processors p is equal to the number of concurrent states the FSM has. The problem size n is not relevant as this is captured already by the number of iterations of this algorithm. In each iteration the problem size is equal to one.

An important property of state machines is that they are inherently sequential. The next state always depends on the current state and the next input character. After evaluating the entire state machine, the final state needs to be communicated back to the host system. When considering state machines, or finite automatrons, we identify two classes: deterministic and non-deterministic state machines.

A deterministic finite automatron (DFAs) is restricted in such a way that it can only be in one state at any given time. Given the earlier metrics, this means that the number of processors is always equal to one. Thus, no strategy exists to execute a DFA in parallel. The best strategy to accelerate evaluation is by processing many DFAs in parallel. This fact plus the fact that a DFA only operates on a graph, an input word and a notion of the current state, all of which are uploaded by the host system, there is no need for any memory allocator.

Non-deterministic finite automatrons (NFAs) can be considered a relaxation of DFAs, in the sense that these state machines can be in several states at the same time. Transitions from state to state do not even require an input character. This divergence could in theory mean that more processors could evaluate a single NFA with a single input string in parallel.

The typical method to evaluate NFAs is converting them to a DFAs. This conversion process introduces many new states consisting of combinations of old states. Applying this method ensures that an input of length n can be processed in n steps by one core. By doing this widely adopted conversion, like the "Level-7 filtering" team did, the same strategy can be followed as for DFAs. This means that no memory allocator is required during the evaluation. For large NFAs this solution is unfortunately not feasible because this might lead to an exponential increase of states, possibly exceeding memory restrictions on the targeted device.

A different way to process an NFA would be keeping track of all current states. This provides interesting possibilities to evaluate a single NFA by several threads in parallel. For a given input character the algorithm could build a work queue of all possible states the NFA could be in. In the next iteration, each work item on this queue can be processed by a separate processor for the next input character, repeating the process until the input word is completely consumed.

The properties of such a work queue are denoted in table 3.2. A work queue will be constructed in every iteration with any size between 1 and the number of possible states s. Freeing of the work items can be done at once. This work queue does not require the use of a memory allocator as it is possible to allocate an array with size s once, and use this same array for every iteration. A malloc routine will however reduce the memory footprint of the execution in the common case, as no overestimation is done.

Evaluating an NFA in parallel by means of a work queue comes with a couple of disadvantages. The biggest disadvantage is the inability to predict how many threads are needed for the next iteration until after the current has finished. In theory the amount of required threads could be any number between 1 and the total number of states. This makes scheduling problematic, as OpenCL requires the number of scheduled threads to be specified before the start of a kernel. Overestimation will waste resources better used by evaluating more NFAs in parallel. Besides this problem, the time required to schedule might also be disproportional to the time spent on the actual evaluation of the state machine. The whole job of the state machine is merely to find the next state(s) in a table based on the input and the current state. Scheduling requires the creation of a new data structure and scanning all states to see if they are a candidate on every iteration.

In order to make a definite statement about these possible disadvantages, an implementation should be created and benchmarked. In-kernel malloc seems to be a prerequisite to creating such a benchmark. Benefits from an in-kernel malloc routine would be code maintainability and the flexibility of acquiring a chunk of memory for each iteration in the process.

3.2.2 Graph Traversal: Graph conversion

Many data sets can be modelled as graphs, like distribution networks, scheduling constraints or instruction dependencies. Graph traversal algorithms are amongst the most commonly used algorithms to process these graphs.

As part of his study, Ate Penders has proposed an OpenCL implementation of an elementary graph analysis problem[38]. One of his initial problems is to convert a list of directed edges into a list of n nodes with their edge-in and edge-out count. Nodes are identified by a unique random integer, not necessarily between 1 and n. This is an example of a graph traversal algorithm that can be used in many applications, for instance in the register allocation algorithm by G. Chaitin[12]. The graph analysis project is aimed at large graphs, possibly with millions of nodes. Additionally, because the nodes are not strictly numbered between 1 and n, the node identifier cannot be used as an array identifier. This restriction requires a node list captured in a data structure more advanced than a linear array.

Investigating the source code of this OpenCL program assumes the following metrics. The problem size n is equal to the number of nodes. The number of iterations i will be defined as the number of edges divided by the number of threads. Finally, p is defined as the number of threads.

As identified, the node list is the only data structure that could benefit from a malloc routine. In order to bypass the lack of such a memory allocator in OpenCL, inefficient hash-table constructions are currently used to accelerate searching of nodes in this list. Worst case these algorithms show no advantage over a simple list iteration as the exact position of a node in a list is not deterministic based on solely its identifier. The developer explicitly requested a memory allocation routine in order to be able to sort the nodes in a binary search tree[5] without a lot of manual labour. This would allow him to use many known optimisations to generate a balanced tree structure[4], reducing the search to any node and insertion of a node based on its identifier from $O(n)$ to $O(logn)$.

The OpenCL kernel would allocate in total n nodes whose size is simply denoted as $sizeof(node)$. Node would be a structure containing an identifier, edge counters, a locking mechanism to avoid concurrent writes and pointers to adjacent nodes. Without going into details, We estimate this to be around 32 bytes for each node.

Although the memory allocations will occur roughly in amounts of $p * i$, as each iteration a part of the processors will allocate memory, there will be no need for freeing parts of the allocated structures separately. Freeing should only be done after the entire tree has been processed and when analysis is done on that tree. This means free only occurs once at the end of the kernel.

A traditional in-kernel memory allocator will allow for the construction of a node tree in a conventional manner, a significant improvement compared to the use of a hash-table. In terms of memory it will however introduce a slight overhead for the administration of the memory allocator compared to the use of a regular array. Furthermore, global access only becomes possible by traversing the search tree, limiting the application of the constructed tree. Predictable indexes might nullify this restriction, and should be considered a useful feature for the designed malloc routine.

3.2.3 Structured grid: Heart Wall

Algorithms that process a multidimensional structured grid of data elements are used commonly for analysis of sampled objects. Key in these algorithms is that the data in each point is altered based on the value of their neighbours. A well-known example is the game of life, but these algorithms are also used more practically for instance in weather forecasting.

These algorithms generally allocate a big chunk of memory which will then be filled with data samples. The grid is then re-used for every iteration of the algorithm, meaning there are no dynamic memory requirements in this part of the algorithm. During execution though, often temporary data needs to be stored as well. In addition, when dividing the grid over several processors each needs its own memory area for storing intermediate results.

The "Heart wall" program investigated was obtained from the Rodinia suite[13]. The program is an implementation of a structured grid algorithm, designed to analyse an AVI video of a mouse heart. Each frame consists of a 2-dimensional grid of pixels, in which the algorithm tracks the movement of the heart. The ideal number of processors p is equal to the number of endo- and epi-points analysed. These points are simply coordinates in the image grid, relevant in medical sense. The iteration count i is defined by the number of frames in the video file. The data size n is equal to the physical number of pixels in a frame.

The first step in this implementation, converting the AVI frame to a bitmap, is done in sequential code. In theory this can be accelerated by using graphics accelerators, a technique for which Cuda offers explicit APIs. However, regardless of the chosen decoding method, the frame decoding will not need to call *malloc()*[46]. In the case of video decoding on Cuda the frames are stored by the decoder in texture memory, outside of the control of the OpenCL program. If video decoding hardware cannot be accessed the decoded frame must be uploaded to the card for the kernel to use.

In the calculations that follow, an array of type "*private_struct*", named *private*[], is allocated on the host system and is partially initialised. This type contains several pointers to floating-point values or arrays of values. Given in the ideal situation the video file is converted to a bitmap in OpenCL, ten of the floating-point values in the *private_struct* array do not need to be shared with the host system. Although these entries are currently allocated separately in a sequential loop, they have been analysed as if they are allocated inside the parallel kernel for the purpose of extracting sane results on the usage of this memory.

The allocated arrays in *private*[] are grids that store intermediate values for the current and surrounding point in the grid. Each of these arrays are allocated and de-allocated exactly once for each processor. The object size is constant and equal to the size of a float. The number of allocated objects is currently constant, but in theory can be limited when taking into account points around the border of an image. Currently each thread would allocate around 1000 floating-point values.

The exact usage of these ten arrays varies. What can be seen is that the individually allocated blocks are all read and written linearly in a part of the algorithm. Some of these arrays are also read in the transposed form (column-wise), a specific type of non-unit stride. One of the arrays, d_tMask, is accessed semi-randomly. Although the pattern is linear, the starting point depends on values found in earlier calculations. It can be interpreted as shifted, but when executed sequentially the access would have more similarities with a random pattern.

Allocating these arrays from inside the kernel will add no improvements to the program. Although it might slightly improve the readability of the code, the overhead of allocating these arrays from within the OpenCL kernel is larger than when done on the host system. This results in an unnecessary performance penalty.

3.2.4 Dense linear algebra: K-means

Dense linear algebra problems generally consist of one or more manipulations done on irregular dense matrices. These matrices are commonly used for representing the availability and weight of edges in a graph, so that an edge from any known source and sink can be found in constant time.

K-means is the problem of partitioning a dataset into k groups or clusters. Each element in the dataset is associated with the partition with the nearest mean. In principle the k-means algorithm works with any data type for which a distance function exists, such as a list of numbers, a set of two-dimensional vectors or a weighted graph. No linear algorithm exists to obtain an exact solution for a k-means problem even with a dataset in the two-dimensional plane[43]. Several efficient heuristics do exist that produce acceptable results.

The K-means implementation we investigated originates from the Rodinia suite[13]. The program randomly picks a center for each cluster. It then calculates the means of each node associated and moves the nodes to different clusters accordingly. From these newly formed clusters a new center is calculated. This process repeats until no nodes move between clusters. We define the number of preferred threads p as the number of nodes in the input graph. The number of iterations is defined as i. The data size n is equal to the number of partitions k.

The implemented program has a very limited OpenMP kernel. The kernel finds for each node in the grid the nearest cluster center, and updates its membership accordingly. The implication of this approach is that none of the temporary arrays is eligible to allocate inside the kernel. All data must be stored throughout the algorithm and thus be transferred for every iteration. For analysis purposes we propose to consider the case where the surrounding for-loop is made part of the kernel. Now three arrays could theoretically be allocated inside the kernel: the membership array, used to store the current associated partition for each node, the new_centers array, used to store the proposed center after each iteration, and two arrays to store the partial results for calculating these new centres.

The membership array is allocated once at the beginning of the algorithm, and freed once at the end. Because the number of objects allocated for each thread is equal to one, this could be replaced by a private variable instead inside a kernel. In the current implementation this is not done because the limited parallel kernel causes this data to be lost after each iteration.

The *new_centers* array also is allocated and freed once. The current implementation is explicitly written to calculate these new centres sequentially, although there is no reason not to use already spawned threads to do this in parallel. Its size, although independent of the number of desired partitions, is constant throughout the calculations and thus not an interesting case for the in-kernel memory allocator.

The same allocation frequency holds for the other two arrays: they are allocated and freed once. They are, however, written to more frequently. Each node in the grid keeps the distance from each cluster in these partial results. These values are therefore written to in a linear fashion, but read on a per-cluster basis. This corresponds roughly with a non-unit stride reading pattern, as also assumed for the heart wall application. What is interesting to note here is that although the kernels could possibly allocate local memory, they must be able to access data from other kernels. This data must therefore reside in global memory, and an indexed array-like structure is preferred here.

As with the structured grid algorithm, values are accessed linearly either when reading or writing. A transposed form of reading is also used, resulting in a non-unit stride access pattern. Allocation and freeing is done only once. The size of the objects is generally equal, but there is one case where a varying number of objects is stored.

A more important similarity with structured grid algorithms is the fact that an OpenCL kernel memory allocator will not be of much use in this program. The data structures are allocated once and predictable in size. Allocating this memory in-kernel would simply introduce an overhead, without giving any real benefit. Also, as we have shown in section 2.4 that global synchronisation is not possible within an OpenCL kernel, our suggestion of extending the kernel with the surrounding for-loop is unfeasible in practice.

3.2.5 Sparse Matrix: Convert to vector

Sparse matrices differ from dense matrices by the large number of zeroes in the dataset. Compression is often applied to reduce the amount of storage required to hold such a matrix, and to reduce the number of arithmetic operations required in certain algorithms. Sparse matrix algorithms are designed to compress these matrices or work with a compressed representation of such a matrix.

The investigated program is SPMV, a benchmark from the Parboil suite implemented in Cuda[2]. The program is designed to convert a compressed sparse matrix in Jagged Diagonal Storage (JDS) format to a vector format. Because the original matrix is stored in a compressed form, there are two ways of measuring the data size. You could either consider the dimensions of the original matrix or the number of non-zero values that are stored in the compressed form. This

implementation iterates over the non-zero data points linearly, thus we define the data size n as the number of such non-zero points. The number of processors p is arbitrary, but ideally equal to this number. The given implementation however is capable of processing considerably more data points in a per-warp block size. The number of iterations i is then $ceil(n/p)$.

This implementation does not require any allocated memory. The output will contain the multiplication of this matrix with a float vector, thus the intermediate result is never relevant. However, when one wants to reproduce or decompress the JDS-compressed matrix, this intermediate result can be stored in memory allocated inside the kernel.

Such a fictive block of memory will be equal in size to the dimensions of the resulting matrix. It will be written to globally and semi-randomly, as the JDS-form relies on restructuring the data values. How it will be read afterwards completely depends on the purpose of the matrix. Allocation of memory to store the data values is done at the beginning of the program and presumably by one thread only, as the data will be used globally and depends on fixed metrics.

Due to this fixed matrix size, this algorithm will not benefit from an in-kernel memory allocator. Again the overhead introduced by such a memory allocator does not come with any advantage.

3.2.6 Spectral: Fast-Fourier Transform

Fast-fourier transform (FFT) algorithms are the most used algorithms in the area of digital signal processing. FFT algorithms are used to convert discrete samples from the time- to the frequency-domain. This allows for efficient analysis and storage of (sampled) analogue signals. In addition these signals can be compressed by leaving sinusoids out, sacrificing precision for a smaller storage size. This is one technique used in lossy audio compression formats such as MP3[37].

For this algorithmic class, we investigated the Cuda implementation of the FFT algorithm found in the Parboil suite[2]. We define the number of data samples as $p * n$, where n is the number of points that will be sampled in a block, and p the number of blocks evaluated. The number of iterations i in this program is equal to 1. Although no memory is dynamically allocated inside the Cuda kernel, there are two candidates that could be if global synchronisation existed in OpenCL: v and d_work.

d_work is used in conjunction with d_source to form a pair of arrays used for pointer flipping. This technique is used often to ensure data does not get overwritten before other threads have read it. Instead of complex synchronisation schemes the program makes use of two data arrays. After an entire iteration all threads synchronise once, and the pointers flip to make the new array the source data for the next iteration. This way source data is available throughout the entire iteration of a loop.

In a GPU-accelerated program this technique is not as effective as in a CPU oriented implementation, because while in the kernel the pointers flip after each iteration, the pointer on the host system to this data is not changed along. This means that the kernel needs to ensure that the eventual data must be available in a predictable place, perhaps even requiring a final memory copy action. For this reason it is possible to allocate d_work on the GPU, given the final output is copied to the d_source array after execution. Instead, this implementation flips the pointers on the host system. This also avoids all the problems described without introducing a large overhead. In this case though, both arrays need to be allocated on the host system.

The other allocated variable is v. This memory block is also used to store intermediate values in the given implementation. Technically, this variable is entirely superfluous when doing pointer flipping inside the kernel, as the intermediate value is only accessed linearly and an earlier value of any entry is never needed after the next step of the iteration. In either implementation, the number of floating point values that need to be stored is constant and available at run time. It can be allocated and freed once respectively at the beginning and the end of the program.

Due to the predictable amount and size of required memory, allocating the memory in-kernel will not give any advantage over allocation by the host. An in-kernel memory allocator will simply introduce unnecessary overhead.

3.2.7 Dynamic programming

Dynamic programming is a technique commonly used to solve problems that can be divided into smaller overlapping sub-problems, where the algorithm effectively uses this overlap to decrease calculation time[28]. Classical examples of problems that can be solved with dynamic programming are the shortest path problem and the 0/1 knapsack problem.

For this class of algorithms, The theoretical working of Dijkstra's algorithm is considered. This algorithm is designed to solve the problem of "finding the shortest path in a graph from source s to sink t"[17]. The algorithm divides the problem by solving for each node n attached to s instead: "what is the shortest path from n to t". This division into sub-problems can be recursively applied until the path from the processed node to t is covered by one edge. The previous sub-problem then becomes trivial as well, as this is the minimum of every node attached to the current. The execution tree obtained then can be traversed back up to end with the shortest path.

This algorithm shows some similarity with NFAs in terms of challenges. Where in an NFA each time step results in a traversal to one or more nodes, the Dijkstra algorithm divides its problem into one or more sub-problems. This corresponds to the traversal to all adjacent nodes.

Depending on the problem, the number of sub-problems may or may not be constant. For the 0/1 knapsack problem for instance the division always results in two paths: "take object n" and "do not take object n". Problems that involve dynamic programming can however always be solved in a linear amount of work, as each node in the constructable graph only needs to calculate its shortest path to the sink once.

Because of the similarities with NFA algorithms, the scheduling strategy to solve dynamic programming problems in parallel roughly corresponds. To design a parallel dynamic programming algorithm a work-queue is required. Upon dividing the work, this queue should be expanded every iteration with all nodes that can be processed in the next iteration, but have not been processed previously. By also keeping track of the iteration number, the back traversal can be done by processing all nodes with the highest iteration number, proceeding to the iteration before until the source has been reached again. Although the size of the work-queue is known in advance, the number of nodes in each iteration is not. Efficiently grouping the nodes per-iteration is a task that is trivial when using a memory allocator, but requires a more complex sorting algorithm when using a static array.

It is important in this case that the work-queue is also used as a stack. First the algorithm will traverse down through the graph to create bundles of work that can be executed simultaneously. After that these bundles should be executed in reverse order. This means that the size of this complete work-queue is dependent only on the size of the graph. However, the size of each bundle might vary and could thus benefit from an in-kernel memory allocator. Moreover, because the number of bundles is unpredictable a simple array allocator is not sufficient here. Ideally it is possible to allocate these bundles as arrays of memory that can be tied together by means of a linked list. Each bundle will be allocated and freed separately, and the positions in this array need to be distributed based on the needs of each thread in the down-traversal phase.

3.2.8 Particle Methods: Barnes-Hut

Particle method algorithms are used to simulate the interaction between any kind of particles, for instance a simulation of molecules in a chemical solution or an N-body simulation, which simulates the influences of gravity between particles. In an N-body simulation, often the Barnes-Hut algorithm is chosen to simplify calculations[6]. By dividing the entire space to a grid of non-uniformly sized squares, each square containing exactly one particle, it becomes computationally easy to disregard the "gravity" of particles far away when calculating the velocity and direction of any particle. With this approach, Barnes-Hut implementations are capable of running in $O(nlogn)$ time, whereas direct particle methods often run in $O(n^2)$. The price is a small and often acceptable loss in precision.

We investigated the implementation created by M. Burtscher et al.[11], a Cuda implementation consisting of multiple kernels. We define the problem size n as the number of particles, the number

of processors p varies between the different kernels. The number of iterations is defined as $i = n/p$, while the number of time steps will be denoted by j.

The algorithm consists of several steps that are executed in order for each iteration. These steps are implemented as different Cuda kernels, to make it possible to vary the thread distribution between them. As this algorithm only updates local data, there is only one data structure that could benefit from a good OpenCL kernel memory allocator: the octree that divides the particles into several squares.

Building such an octree will benefit from using a malloc routine, as the number of nodes in the octree is not known in advance and may even vary between iterations. In general the octree contains at least as many nodes as there are particles in the system, but if the splitting of one square leads to an empty new square there might be more nodes than particles. The ability of allocating memory from inside a kernel will make the tree creation transparent and allows for parallel execution.

What the implementers have decided to do instead is use a global array that is managed by one or more atomic operations. Each time step the same array is reset and re-used, corresponding to a single free for all the memory previously allocated. Usage of this memory is done by following pointers, leading to a random access pattern. An attempt at improving locality by reordering is implemented, but the access pattern remains unpredictable and random.

This particular program already uses a heap allocator inside the OpenCL kernel, as the atomic operations used on the global array resemble the operations performed by a simple memory allocator. By using a true dynamic memory allocator, the Barnes-hut implementation will lose the atomic operations on the global array and thus benefit from a higher maintainability of the code.

3.2.9 Unstructured grid: Back propagation

The "unstructured grid" class of algorithms is used to evaluate grids of irregular layout. One algorithm that uses an unstructured grid is the back propagation algorithm. This algorithm is used in the field of artificial intelligence to train a neural network[30]. Each node in a neural network consists simply of a "threshold" value. Any number of inputs are converted to a single output considering the threshold of that node. The training step simply consists of calculating the error of each node, and adjusting the threshold accordingly. Both are done in parallel for each node.

We investigated the back propagation implementation found in the Parboil benchmark suite. The number of processors p is equal to the number of nodes in the grid. The number of iterations i is arbitrary and equal to the number of training rounds. The problem size is equal to the number of processors.

The given implementation only executes small kernels in parallel instead of evaluating the entire neural network. For OpenMP this is considered good practice, as the control logic only needs to be executed once. On a GPU application however this forces the use of the global memory space for variables. Unfortunately, because there is no way of synchronising globally in OpenCL without returning to the host system, this is the only way of representing a back-propagating algorithm.

The implication of this observation is that in the back propagation algorithm, we found no use-case for an OpenCL kernel memory allocator. Different specific implementations of unstructured grid algorithms might have intermediate values, but it has been shown earlier that these values can be stored in pre-allocated memory or local variables more efficiently than in kernel-allocated global memory when the layout and size of the grid does not change between iterations. In unstructured grid algorithms this is indeed the case.

3.2.10 C++

One use-case not covered by the list of algorithm classes, but an important one nonetheless, is the C++ programming language. This language is characterised by its object oriented nature. This means that objects, a combination of properties and methods to manipulate them, can be created and deleted on the fly by using the *new* statement. In C++, the new-statement for an object is

often implemented by a call to the available memory allocator to request space for this object, after which the object is initialised.

AMD developed the "OpenCL Static C++ kernel language extension" as part of their AMD APP SDK[15], a subset of the C++ language. Lacking a memory allocator to handle the *new* statement, this specification currently only covers static objects.

Dynamically creating an object requires a piece of memory large enough to hold the data section of this object. This requires a flexible memory system capable of allocating and freeing objects as the developer pleases. Contrary to earlier use-cases, there is nothing to conclude about the pattern in which these objects get allocated and freed. This is entirely up to the developer, which means a very flexible memory system must be available. In addition, there might be need to extend this C++ subset to allow for parallel object creation. Research on how this could be exposed and aid in making the process of allocating more efficient is out of the scope of this project, but acknowledging that object creation is a use-case for a memory allocator does bring a whole new set of constraints to the design.

3.3 Conclusions

After investigating a set of parallel programs covering the majority of the different algorithm classes identified by K. Asanovic et al.[3], a couple of conclusions can be drawn on the usage of malloc in parallel applications.

First of all, it seems that even in architectures that offer in-kernel memory allocation, all memory is allocated at the beginning of, or actually before, the execution of a kernel. This memory is freed after finishing the execution of all work. The mix of pre-allocated blocks of memory for intermediate or global data and variables for local data appears to be sufficient for any program that implements one of the investigated algorithmic classes.

Yet we have found several cases where the use of an in-kernel malloc routine, even with the current limitations in GPGPU, would be beneficial to the application. The situations recognised were mostly constructing a non-linear data structure such as a tree or a variable sized work queue. These problems can be avoided by careful programming, but often programmers end up writing their own memory "allocation" code, re-inventing a heap allocator over and over again.

Based on the found results, the demands for a memory allocator seem to be slightly different from those of a sequential architecture. Most importantly, memory allocated by different threads will be freed all at once. There seems to be no need to free every single allocated block independently, but rather entire data structures would be discarded at once. Note that this does not mean the entire heap must be cleared at once, as it can be used to allocate more than one data structure. This observation allows for an efficient freeing algorithm, resulting in considerably less overhead. It must be kept in mind though that this is a one-time win, as freeing data is not done often anyway.

The second observation we made is that the allocated structures consist of large amounts of elements of the same size and structure. This might be advantageous when trying to meet the demand of global access to this data structure. Global access becomes a lot easier when pointers can be used roughly like indexes in an array, and having regular sized objects greatly helps to achieve such a goal.

Based on the samples it is also safe to conclude that threads will write to their own chunk of memory either linearly or randomly. However, when at least one thread requests memory for more than one element, even the applications with threads writing to their memory linearly will produce a random access pattern for the global system. There is no solution yet for the memory allocator to turn this into a more local access pattern without introducing extra memory reads.

A big exception to the found patterns is the case of C++ dynamic object creation. In this case, there is a definite need for a memory allocator to store these objects, but the use-cases for these are endless and cannot be generalised.

Chapter 4

Allocator design

So far we have showed several uses for a memory allocator in OpenCL programs. In order to facilitate dynamic memory allocation for future programmers we propose a design for a memory allocator in this section of the paper. First the identified constraints are listed, after which the heap allocators APIs is presented.

4.1 Constraints

In order to define the boundaries in which a memory allocator must operate, we have identified constraints from a number of sources. First of all, there are some general requirements that every parallel code and every memory allocator needs to meet. Then, there are some additional constraints enforced by the OpenCL environment and the platforms targeted by OpenCL. Finally, there are some requirements that follow from the use-case survey we presented in chapter 3.

4.1.1 General requirements

Low fragmentation Every memory allocator suffers from a certain amount of fragmentation, be it internal- or external fragmentation[40]. As explained earlier in section 2.5, the DLMalloc algorithm suffers from external fragmentation because the size of the free blocks do not always perfectly match a programs memory demands. Hoards algorithm is not as much susceptible to this external fragmentation by making use of superblocks, but might cause more internal fragmentation because it uses a superblock-based system with size-classes which might not perfectly match the memory demands of a program. Also, having many underutilised superblocks can be considered a form of fragmentation.

High speed Reducing fragmentation cannot come at any price. As we are discussing platforms that strive for high performance, some memory fragmentation might be considered acceptable if it leads to a bigger performance. In any case, the memory manager should not be the bottleneck in an application.

Ease of use The goal of any library is to facilitate a complex feature for programmers, in such a way that the programmer does not have to worry about any implementation details of this feature. Although a new complex memory allocator might map perfectly to current hardware, this should not lead to an API that is difficult to understand. If this heap allocator library should be a tool to enable developers to experiment more easily with OpenCL, it should be designed with ease of use in mind.

4.1.2 Platform requirements

Thread safety Although all software should be either single threaded or thread safe, no risks can be taken with GPGPU. The probability of any fault occurring, caused by a race condition error in the programming code, grows with the number of threads executed in parallel. Because GPUs feature many cores each processing a work-item (thread) of the same program, extra care should be taken in avoiding race conditions.

Scalable In the coming years GPUs will likely evolve in two ways: removing bottlenecks that are no longer acceptable for GPGPU and adding more cores. The latter trend means that a memory allocator should scale extremely well with the number of work-items and cores. Having a memory allocator whose performance decreases superlinearly with number of used work-items will certainly be unacceptable for massive-parallel platforms.

No GPU-to-host communication As explained in section 2.3, there currently is way for an OpenCL kernel to communicate with the host system. In the case of memory allocation this communication channel is desired to request pages of memory from the host system that can be further distributed by the in-kernel heap allocator. As a result it is impossible to build a memory allocator in OpenCL with the same functionality as heap managers in C libraries adhering to the POSIX standard.

The solution NVIDIA takes to overcome this problem is to allocate a fixed heap of 8MB that is used by the kernels memory allocator[36]. If required, the programmer can change this heap size arbitrarily before starting the Cuda kernel. This means that this heap allocator does not solve the problem of memory usage overestimation, introduced when the amount of required memory is not known exactly before execution. Since we cannot alter hardware or firmware to facilitate the desired communication, we have to follow a similar strategy in our allocator.

4.1.3 User requirements

Generic Although the XMalloc strategy, described in section 2.5, for reducing the number of effective allocations is quite efficient, we cannot enforce synchronised allocation in all cases. C++ is a good example of a use-case that cannot be addressed by XMalloc. Developers should be able to use a thread-safe generic allocation method that does not enforce synchronisation of work-items.

4.2 Technical design

Based on these constraints, we propose a layered interface. Layered memory allocators have been discussed by E. Berger et al.[8], concluding that the benefit of maintainable and easy-to-read code resulting from a layered design far outweighs any possible benefits a single-layer custom-optimised memory allocator might have. With a correct design the performance impact can even be negligible, while it is a lot easier to find bugs and identify areas of improvement. We therefore designed a heap allocator consisting of two layers.

At the bottom, we propose the "low-level" heap allocator. This component behaves roughly similar to the existing memory allocator in libc[1]. The top layer should be an "object" implementing a basic data structure that uses the bottom layer heap allocator to fulfil its memory needs. Depending on the specific details of such a data structure, usage of the low-level allocator may be optimised.

To demonstrate the concept, we implement a prototype top-layer called "ArrayList". This object will implement a list of equally-sized objects which are globally accessible, inspired by the ArrayList object available in Java[1]. Clearing of the list can be done with one function call, and making the list grow can be done either per work-item, or per work-group. The latter grow function applies the strategy proposed in [23]. Programmers are able to use either the low level allocator or

[1] http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html

the ArrayList, but are also free to create their own data structure on top of the low-level memory allocator.

This two-layer design has a number of benefits. The low level memory allocator is close to what traditional programmers are used to. This similarity could attract new programmers to make early prototypes without having to worry about the complex memory model of OpenCL straight away. More importantly, it does not enforce any form of synchronised access, meaning that this low-level heap is usable for every allocation case.

The ArrayList object demonstrates how we can offer more performance to programs that meet the synchronisation demand of the prefix-sum reduction algorithm. Using the low-level heap allocator is not restricted to just the ArrayList object; different list-type objects could be created that use the same malloc back-end. If later it turns out the current malloc back-end is not very efficient, another back-end that implements the same APIs can easily be dropped into place. In short: such a two level design embraces all the functionality we feel a programmer desires without limiting the programmer to do things differently if the proposed implementation does not completely suit his use-case.

4.2.1 Low level heap allocator

Heap manager
+clSBMalloc_init(dev:cl_device_id,ctx:cl_context, cq:cl_command_queue,prg:cl_program, sb:unsigned int): cl_mem
+clheap_init(heap:__global struct clheap *) +malloc(heap:__global struct clheap *,size:size_t): uintptr_t +free(heap:__global struct clheap *,block:uintptr_t): void

Figure 4.1: Technical Design: Heap manager

The heap managers API is shown in figure 4.1. On the host side it consists of one function:

clSBMalloc_create(cl_device_id dev, cl_context ctx, cl_command_queue cq, cl_program prg, unsigned int sbs) - Create a heap on the host and the device side. This heap is a buffer containing its state followed by *sbs* superblocks. Each superblock is of a fixed size, defined as a constant in clSBMalloc.h. This function will execute the *clheap_init()* OpenCL kernel to set up all the data structures, and returns a *cl_mem* object pointing to the entire heap. This *cl_mem* object should be passed as a kernel parameter to any kernel which requires the use of the heap.

On the device side the programmer can now use the following API:

clheap_init(__global struct clheap *heap) - Initialise the heap, making sure the begin and end-pointers are valid GPU pointers. Due to the lack of global scope variables or the ability to define a pointer in a constant or macro in OpenCL, *heap* will remain a parameter for all malloc-related functions.

malloc(__global struct clheap *heap, size_t size) - Allocate *size* bytes from the heap pointed to at **heap*. Returns a *uintptr_t* pointing to the newly allocated block, or NULL on failure. When out of memory, the behaviour is undefined, but could result in an endless loop.

free(__global struct clheap *heap, uintptr_t *block) - Free *block*, marking it allocatable by any work-item.

4.2.2 ArrayList

ArrayList
+clArrayList_create(dev:cl_device_id,ctx:cl_context, cq:cl_command_queue,prg:cl_program, objsize:size_t,heap:cl_mem): cl_mem
+clArrayList_init(l:clArrayList __global *, size:size_t,heap:struct clheap __global *): void +clArrayList_get(l:clArrayList __global *, index:uint): uintptr_t +clArrayList_clear(l:clArrayList __global *): void +clArrayList_grow(l:clArrayList __global *, items:size_t): uintptr_t +clArrayList_grow_local(l:clArrayList __global *, items:uint,rMem:size_t __local[]): uintptr_t

Figure 4.2: Technical Design: ArrayList

In figure 4.2 the API of the ArrayList object is given. On the host side it consists of a single function:

clArrayList_create(cl_device_id dev, cl_context ctx, cl_command_queue cq, cl_program prg, cl_uint objsize, cl_mem heap) - This function creates and sets up an ArrayList data structure. Set-up is done by calling the *clArrayList_init()* kernel on the device. This ArrayList is configured to contain objects of *objsize* bytes. The last parameter is a pointer to the heap as returned by *clSBMalloc_create()*. All objects inside the ArrayList will be allocated from this heap. This function returns a *cl_mem* object pointing to the ArrayList.

On the device side the API consists of the following functions:

clArrayList_init(clArrayList __global *l, size_t size, struct clheap __global *heap) - Initialise the ArrayList, making it contain objects of *size* bytes. Any required memory will be allocated from the heap.

clArrayList_grow(clArrayList __global *l, unsigned int objs) - Grow the given clArrayList with *objs* blocks of the size given on initialisation. Returned is a pointer to the allocated block.

clArrayList_grow_local(clArrayList __global *l, unsigned int objs) - Grow the given clArrayList with *objs* blocks of the size given at initialisation. This function needs to be called by every work-item in the work-group, as it performs a local synchronised reduction to gather the total memory requirements. If not all work-items enter this function because branches have diverged, the system will lock up. If this function ends it returns a pointer to the first object for this work-item.

clArrayList_clear(clArrayList __global *l) - Clear the entire ArrayList, freeing all allocated blocks inside.

clArrayList_get(clArrayList __global *l, size_t i) - Get an arbitrary object. This function returns a pointer to the element at the *i*'th location.

Because there is no way of doing global synchronisation without interrupting the program, there is no API for doing a global prefix-sum reduction amongst all threads in the system.

Chapter 5

Implementation

After defining the APIs, there are a lot of implementation details that need to be decided. These decisions are based on the requirements we identified and are the product of several iterations of the design and implementation phase. In this chapter the implementation decisions are documented.

5.1 Low level: heap manager

At the base level of KMA lies the heap manager. The heap manager is schematically represented in figure 5.1.

The administration of KMA is inspired by Hoard[7], managing superblocks consisting of equally-sized blocks. The heap itself is a single read-write buffer allocated by the host system. This buffer contains the state of the heap, followed by a series of superblocks. Similar to XMalloc[23], KMA only manages a single global heap. Figure 5.2 contains the structure of the *heap*, consisting of the *heap state* and the *superblocks* used by KMA, along with the relation between these entities.

The heap state section contains several structures and values that together form the current state of the heap. *bytes* is an integer containing the total size of this heap in bytes. This value is used on initialisation of the heap. *free* is a queue object containing a singly-linked list of free superblocks. The *sb* array is a hashmap containing a slot each possible block size, meant to store a pointer to an active superblock. Each entry in this hashmap can point to at most one active superblock, like illustrated in figure 5.2.

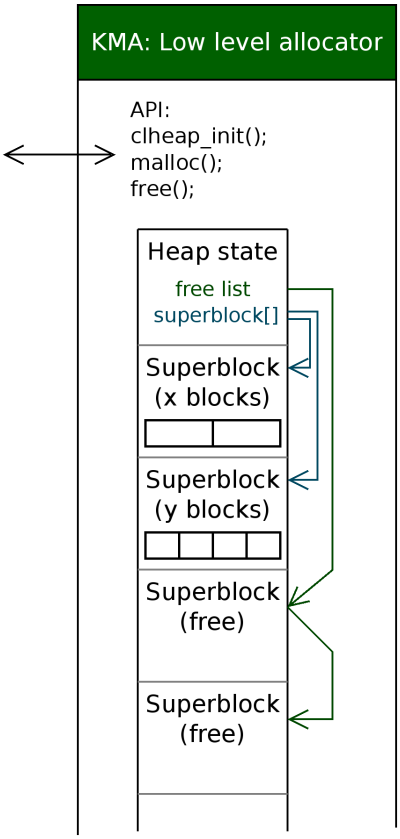

Figure 5.1: Heap structure

5.1.1 Blocks and superblocks

In KMA, a superblock can be in one of three different states: active, inactive or free. Active and inactive superblocks contain at least one allocated block. Free superblocks contain no allocated blocks, and are linked together in the *free*-list.

Because a superblocks next-pointer must exist even after it is removed from the free-list, superblocks cannot be safely coalesced. The size of an allocated block is thus

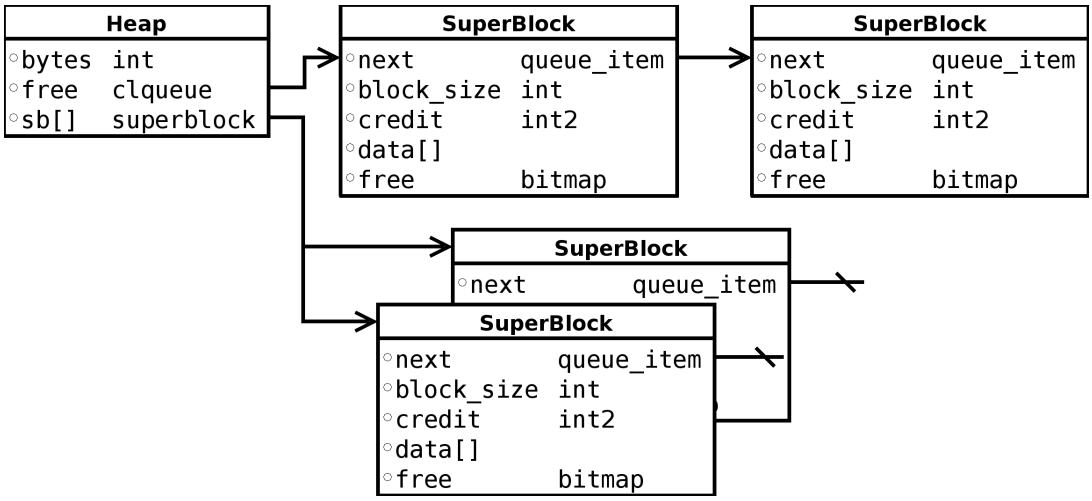

Figure 5.2: Superblock administration

upper bound by the size of a superblock minus its header and footer. The possible sizes of blocks inside this superblock are approximated by:

$$\{2^n \quad | \quad 2 \leq n < \log(\text{superblock\_size})\} \tag{5.1}$$

Heuristics are used to determine the actual block size. These heuristics are designed to minimise the amount of space wasted on almost-fitting large blocks.

Each superblock contains a header consisting of a few properties. *next* is a pointer to another superblock, which can be used by the free-list. *block_size* contains the size of the blocks in this superblock. *credit* contains the number of slots and the number of free blocks. These header values are followed by the data segment containing the actual blocks. Every superblock ends with a bit-map indicating which blocks in this superblock are taken and which are free.

The life-cycle of a superblock looks as follows: when a new superblock is required, one is taken from the *free*-list. The header and bit-map of this superblock are initialised and the superblock is stored in the right entry of the *sb* list in the heap state. At this point the superblock is active. Subsequent allocations of similarly-sized blocks will be done from this superblock until all its blocks have been allocated. As soon as all the blocks in this superblock are allocated, it is removed from the *sb* list, making this superblock inactive. Freeing all blocks inside an active or inactive superblock causes it to be reconnected to the *free*-list, allowing the free superblock to be re-used later on.

Block size heuristics As we mentioned, the exact block size is not precisely a power-of-two. A heuristic is used to determine the possible block size and number of blocks within a superblock, depending on the requested amount of memory. This heuristic avoids fragmentation between the end of the last block and the start of the bitmap. For requested blocks of size $\sqrt{(superblock\_size)}$ or smaller the following steps are executed to find the parameters of a superblock:

1. Determine the size of a block s, being the next power of two
2. Determine the number of blocks: $c = \frac{data\_size}{s}$
3. Determine the size of the bitmap in bytes: $s\_bm = ceil(\frac{c}{32}) * 4$
4. Correct the number of blocks to leave room for the bitmap: $c = c - ceil(\frac{s\_bm}{s})$

For requested blocks larger than $\sqrt{(superblock\_size)}$, a slightly different algorithm is used:

1. Determine the size of a block s, being the next power of two
2. Determine the number of blocks: $c = \frac{data\_size}{s}$
3. Determine the size of the bitmap in bytes: $s\_bm = ceil(\frac{c}{32}) * 4$
4. Correct the size of a block to fit: $s = s - ceil(\frac{s\_bm}{c})$
5. Round the size of a block down to the previous multiple of 4 bytes: $s = s\&NOT(0x3)$

5.1.2 Free-list

At the heart of any memory allocator lies its administration of free memory blocks. By organising this free memory administration properly, a lot of decisions could be simplified resulting in faster allocation algorithms. In essence this administration is nothing more than one or more lists of free blocks. Depending on the chosen data structure, there is one or more lists that could be ordered or not.

In section 2.4 we have briefly explained why locking algorithms are unusable in OpenCL software. Taking this into account, lock-free algorithms for its data structures must be used for building a memory allocator in OpenCL C.

Ideally lists of unknown lengths are used to keep flexibility in the size of the heap. The most common way to implement such a list is called a "linked-list". This data structure stores in each element a pointer to the next, linking different memory blocks together.

Linked List Of the linked list type there are two implementations: singly-linked-lists and doubly-linked-lists. Both lists consist of nodes that contain a pointer to the next item in the list. The doubly-linked list uses some additional memory to store a pointer to the previous entry in the list. Both lists should support the "insert" and "delete" operation, where a doubly-linked list can execute the delete operation in less time because it does not need to search for the node pointing at the to-be-deleted node. These data structures are usable in sequential environments and parallel environments that allow for locking: When considering unordered linked lists, both take $O(n)$ time for insertion. A singly-linked list takes $O(n)$ time for deletion, and a doubly-linked list requires $O(1)$ time when deleting a node. Locks are used to prevent elements from being deleted that are used by other threads. Because the overhead in memory for a doubly-linked list is small and can even overlap the data-section of a memory block this is an ideal structure for memory allocators.

A lock-free implementation of a doubly-linked-list is documented by H. Sundell et al.[41] This algorithm uses CAS to link and unlink nodes in several steps. In this process it assumes that the "forward" list always contains all connected nodes whereas the backward list only contains hints about where the previous node is supposed to be. This previous pointer can be used to determine the actual previous node without having to traverse the entire list.

One particular problem this algorithm tackles is related to traversing the list. For this traversal, every thread traversing the DLL has a cursor pointing to the node that it is currently using. Problems arise when a thread deletes a node, as it could invalidate the cursors of other threads without knowing so. H. Sundell et al. propose to place these cursors in global memory, and alter all of them as part of the deletion process. This way, other threads no longer risk pointing to something not a node in the list.

Although this solution is effective, we consider this solution not viable for GPGPU. The large amount of work-items might cause the deleting process to spend a lot of time on updating the cursors. These cursors either need to be pre-distributed to the threads or allocated every time a work-item wants to traverse the linked-list. Having one cursor in memory for each work-item means a lot of memory is required. Allocating cursors instead of pre-distributing them to lower

the amount of required global memory requires an allocation algorithm, which is precisely the problem we are trying to tackle in the first place. To make matters worse, all traversal operations will be further slowed down by the fact that this cursor is no longer stored in a local register, but rather in much slower global memory.

Queue Because it is not desirable to update the cursors of other threads, we rather consider data structures that do not require traversal of the entire list and thus do not require a cursor per work-item. There are two obvious data structures that fit this profile: queues and stacks. Both can be implemented using the same technique as a singly-linked list, but only the first and/or last node in these structures are relevant. Using a singly-linked list has the additional benefit of ending up with a simpler lock-free algorithm; there are less pointers to update on insertion and deletion and thus the invariant becomes less complex. For KMA, we chose to use a queue for our free list.

The downside of a queue is that it is not an ordered structure, so all nodes in this queue must be equal. For our memory allocator this means that all enqueued superblocks must be equal size. Earlier we already explained that superblocks cannot be coalesced because its pointers must remain valid even after dequeuing. These two facts combined lead up to the size restriction we explained earlier.

The chosen queue implementation must meet a few demands. Most importantly it cannot rely on allocation, because that is the the problem we are trying to solve in the first place. Secondly both enqueuing and dequeuing must take immediate effect. Deletion of nodes cannot be delayed until all references have been solved, because the entire object is required by the memory allocator and must be returned on deallocation.

A simple and practical algorithm with these properties is a small alteration of the one described by Michael Maged et al.[32]. This algorithm is lock-free and correct as proven in [21]. Moreover, it is a simple solution that utilises nothing more than just three different pointers: head, tail, and the next-pointer in each enqueued item. The details of our implementation are shown in section 6.1.1, along with a static performance analysis.

Our queue implementation circumvents any ABA problems by applying the solution proposed by M. Maged et al.[32], like explained in section 2.1. It uses pointers that consist of a 20 bit address and an 11-bit tag counter. Because enqueued blocks are all aligned and equally sized, the address is encoded as an index. Decoding this index is done by multiplying its value by the superblock size and then adding the address of the first superblock to the result. With 4KB superblocks we can manage a heap of up to 4GB with 32-bit atomic operations with this addressing scheme, much more than we expect to require.

When a queue runs empty, it could be possible that both the head and tail pointer need to be updated when enqueueing or dequeueing an item. This is a problem because a CAS operation can only update one value at the time. The algorithm proposed by M. Maged avoids this problem by initialising the queue with a dummy object and never allowing the queue to run completely empty. To avoid wasting a complete object on the queue, the dequeue operation returns not the data of the item it dequeues, but the data of the first object after this item.

In our case, this strategy is infeasible because the queue does not contain data, but actual memory blocks. Instead, our implementation of the queue does not add a dummy node on initialisation. The dequeue operation returns a pointer to the just-dequeued block and always keeps one real superblock enqueued. Our implementation thus takes the loss of one superblock of memory for granted for the sake of a simple and correct algorithm.

5.1.3 Algorithms

As seen in the API documented in chapter 4, KMAs low-level memory allocator implements three API calls: *clheap_init()*, *malloc()* and *free()*.

clheap_init() This kernel is used solely to initialise all the data structures of the heap. It will first set the state of the heap to sane default values, and then enqueue all the superblocks from

the read-write buffer into the free-list. This kernel is not thread-safe and should always be called from the host system with only one work-item.

malloc() The malloc routine is used to allocate one block. The code performs this task in three steps, as illustrated by the code snippet below.

```
void __global *
malloc(__global struct clheap *heap, size_t size)
{
        int size_class;
        unsigned int slot,i;
        struct clSuperBlock __global *sb;

        size_class = _clSBMalloc_sbid_by_size(size);
        if(size_class < 0)
                return NULL;

        sb = _clSBMalloc_reserve_block(heap, size_class, &slot);
        if(!sb) {
                return NULL;
        }

        return _clSBMalloc_get_block(sb, slot);
}
```

First the malloc routine obtains the right size-class for the block requested. If no such size-class exists, for instance when the requested block is larger than the size of a superblock, a null-pointer is returned indicating failure.

When the size-class is known, the routine tries to reserve a block in a superblock of this size class by calling *_clSBMalloc_reserve_block*. This routine does two things. First it finds a suitable superblock in the *sb* hashmap. If the desired size-class does not have an active superblock, one is taken from the free-list, initialised and added to the hashmap. Once this is done, the credit of this superblock is decreased by one atomically by using the CAS operation. This process is repeated until the CAS operation succeeds. If the credit is zero, the superblock is detached from the hashmap making it inactive. This routine returns a pointer to the superblock in which a block is reserved. *Slot* is set the old credit.

If reservation fails, for instance if there are no more available superblocks, a null-pointer is returned. Otherwise, *_clSBMalloc_get_block* is called to obtain a block from the superblock returned. This routine iterates over the superblocks bitmap, which are used to indicate which blocks are taken. If a zero-bit is found, this bit is set using the atomic OR operation. Upon success, a pointer to the corresponding block is returned. On failure, iteration continues from that point. The *slot* value is used as a starting point for iteration, to avoid that the routine always starts its iteration at the same point and iterates over a long list of taken blocks.

free() When the free-routine is called with the address of a block, the opposite is done from *malloc*. First the credit of the superblock that contains the deallocated block is incremented by one. If the credit ends up being equal to the total number of blocks in this superblocks, credit is set to 0 instead to prevent further allocation from this superblock. Then the blocks taken bit in the bitmap is unset using an atomic AND operation. Finally, if this free-operation results in a completely free superblock, the superblock is detached from the *sb* hashmap and enqueued to the *free-list*.

5.2 Low level: "Poormans"-heap

For certain benchmarks, we want to compare the performance of KMA with performance of a situation without any heap allocator overhead. As the use-cases we target cannot run without a heap allocator, we implemented the next best thing: the "poormans"-heap. Its concept is coined by Doug Lea[29], who describes it as an extreme example of a fast memory allocator which is hardly ever acceptable due to its memory wastage. In our implementation, this simplified heap has an interface identical to KMAs low-level heap allocator. This implementation is not a complete heap because freeing a block of memory does not lead to re-using it.

The poormans-heap consists of a big buffer containing the state of the heap, followed by the heaps data segment. The state of the heap consists of a *base* pointer, a *head* offset and an integer containing the size of the total heap in bytes.

5.2.1 Algorithms

As the free routine cannot be implemented, it just returns *true*. Malloc is implemented as follows:

```
void __global *
malloc(struct clheap __global *heap, size_t size)
{
        uint32_t ret, sz = size;

        ret = atom_add(&heap->head, sz);
        if((ret + size) > heap->tail)
                return NULL;

        return (void __global *)&heap->base[ret];
}
```

As we see in this code snippet, malloc simply increments the heaps *head*-pointer with the size of the requested block. Upon success, the old *head*-pointer is returned. When the new *head*-pointer was incremented to a value beyond the tail of the data segment, a null-pointer is returned indicating failure. For benchmark purposes this implementation is sufficiently correct, although technically the *head*-pointer could wrap around, leading to unpredictable behaviour. This issue can be solved by restoring the *head*-pointer to its old value when it is detected that the new value is larger than the heaps *tail*.

5.3 High level: ArrayList

Keeping a consistent heap state is expensive; often lock-free algorithms have no upper bound in execution as they consist of an infinite loop that returns on success. An analysis of the malloc algorithm in KMA-1 can be found in chapter 6.1.1. A good way of minimising the overhead and improving speed is reducing the need of this memory allocation routine. The XMalloc paper[23] gives a good strategy for this.

ArrayList is a proof-of-concept top layer that utilises the concept of the XMalloc approach of using prefix-sum reduction to reduce the amount of calls to malloc. ArrayList implements a list of equally sized objects, allocated from the heap. Upon growth of the ArrayList, one or more blocks are allocated by calling the low-level *malloc()*-routine. The allocated blocks of memory are connected in a singly-linked list. Each allocated block starts with a header containing information about the number of objects inside this block. This design makes it possible to do indexed access to arbitrary blocks inside the ArrayList, regardless of which thread allocated it.

5.3.1 Reduction algorithm

As explained in section 2.5, the strategy behind XMalloc relies on doing a prefix-sum reduction to allocate memory for all work-items in a work-group with one call to *malloc*, and distributing the returned block of memory over all work-items. This prefix-sum reduction algorithm works as follows: given a one-dimensional projection of a list, obtain for each element the sum of all elements earlier in the list. An example execution of this algorithm for four arbitrary numbers is given in figure 5.3

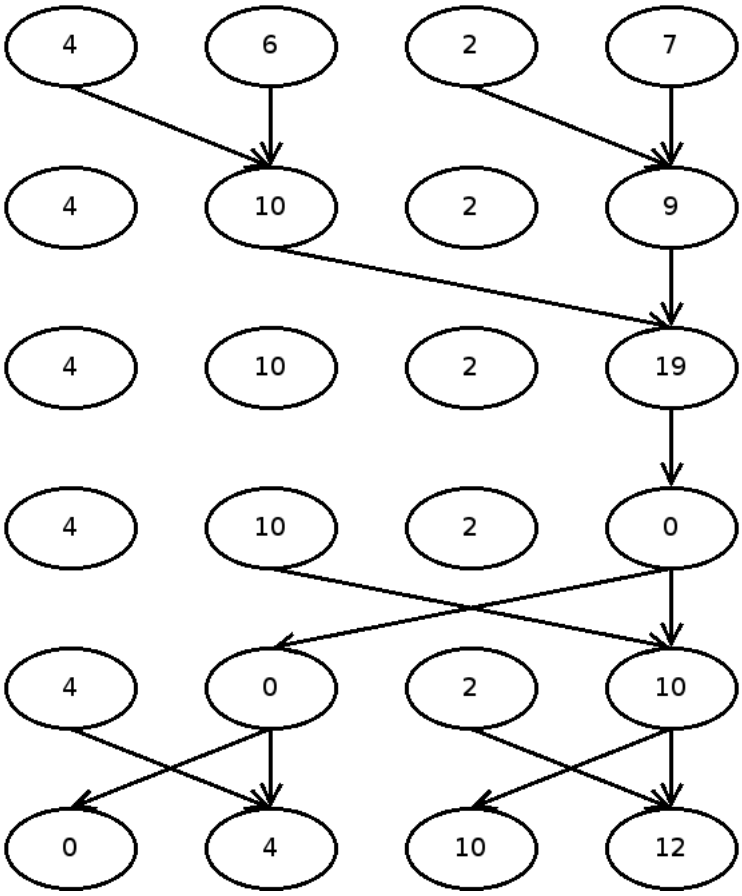

Figure 5.3: Prefix-sum reduction algorithm

The prefix-sum reduction always takes $2 * log(n) + 1$ steps to complete, where after each step all cores must synchronise. The reduction consists of two phases. First there is the "up-sweep". In this phase partial sums are calculated, along with the total sum of all values. The second phase is the "down-sweep", where careful swapping and addition of these partial sums leads to the correct prefix-sums. The time complexity of this algorithm is $O(log(n))$. Because after the up-sweep the total of all values is known, no separate algorithm is required to calculate the total.

This prefix-sum is used to distribute memory within a single block over multiple threads. In this example, if a memory block $M[0..18]$ is allocated, the second "thread" is expected to skip 4 elements, be it bytes, words or any other object size, and is thus allocated the memory chunk $M[4..9]$. The chunk that the third thread can use starts at an offset of 10. By not setting the last item after the up-sweep to 0, but rather to the memory address of the start of the memory block, the correct pointers are distributed in the down-sweep instead of the offsets.

A prefix-sum addition of size n only requires $n/2$ threads to process. By padding the dataset with enough zeroes to make its size equal to the next power of two, no special corner cases need

to be considered when the size of the dataset is not a power of two. The implementation of prefix-sum in our ArrayList thus works with work-groups that contain a non-power-of-two number of work-items.

5.3.2 Possible variations

The solution we implemented in ArrayList is like proposed by X. Huang et al[23]. It is designed to minimise the memory usage while improving performance compared to per-thread allocation, under the assumption that each thread in a work-group allocates at roughly the same moment. Different demands could lead to different design decisions though. We propose some alternatives in the next paragraphs.

Local versus global synchronisation As explained in section 2.4, OpenCL 1.2 does not specify a mechanism for global synchronisation. If this comes available it would be interesting to compare the currently implemented per-work-group prefix-sum reduction allocation to an implementation that applies the prefix-sum reduction on all work-items executing this kernel.

As an obvious advantage, global prefix-sum reduction will result in even less calls to the underlying *malloc()*-routine. Instead of calling *malloc()* once for each work-group, it will only be called once in total. Depending on the used heap allocator this may or may not give a significant performance improvement.

Although this sounds a lot more efficient, there are quite a few caveats. First of all running the prefix-sum locally per work-group greatly reduces the overhead for synchronisation because there are less threads to involved. On a GPU several work-items in a work-group share a program counter, meaning these work-items are already synchronised for free. Global synchronisation, if even possible, means all work-groups should be synchronised, wasting valuable time on waiting for other work-groups.

Secondly a global prefix-sum algorithm cannot use local memory for the entire reduction, and instead requires the use of global memory for most steps in the global prefix-sum reduction. Global memory is a lot slower to access than local memory, so performance of the prefix-sum algorithm will degrade.

Finally both the work-complexity and time-complexity increase when doing a global prefix-sum reduction. Normally the prefix-sum algorithm runs in $O(log(n))$ time and processes $O(nlog(n))$ work for n work-items . In the case of a per-work-group prefix-sum reduction between w work-items, the algorithm will execute in $O(log(w))$ time and process $O(nlog(w))$ work. With any GPU design the work-group size $w =< n$, so doing a locally synchronised prefix-sum is always more efficient from the algorithms point of view.

Which of the two approaches is more efficient depends on the performance and scalability of the used back-end, and experiments should show whether a global synchronised prefix-sum is a desired feature.

Max distribution Another interesting variation on the XMalloc approach is using a different reduction algorithm. Before explaining why prefix-sum might not be the best choice it is important to consider the properties of the different use-cases mentioned in section 3. Table 3.2 contains a column called access patters. The patterns here are explained in [26]. The “access pattern” column in the table lists the local access pattern for a work-item.

Consider any variable with a linear local access pattern. The following example will assume three work-items: the first work-item requests three blocks of memory, the second requests one block and the third requests two. After allocation, the following global access will be done on read or write operations when the prefix-sum distribution method is chosen:

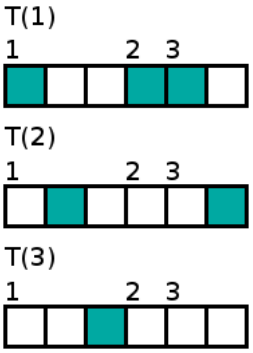

Figure 5.4: Implementation: Prefix-sum distributed global memory access pattern

In figure 5.4 you can observe that even though the different work-items access their own memory linearly, the global memory access pattern is described best as being "random". GPUs benefit greatly from having a more linear global access pattern, because they are optimised for transferring large adjacent blocks of memory. If the accessed memory is fragmented, memory burst-modes will not work, leading to degraded performance. An ideal distribution will thus look like following.

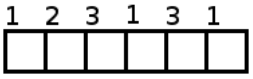

Figure 5.5: Implementation: Ideal global memory access pattern

Although this ideal case would greatly improve the memory bus usage, it cannot be achieved in practice. The main advantage of the prefix-sum algorithm is that the offset for accessing each object can easily be calculated by using only data local to the work-item and the start address as distributed by the prefix-sum. To get to an ideal pattern like in figure 5.5 an algorithm must be used that distributes more information. Probably a pointer table needs to be used to achieve a memory distribution like depicted in figure **??**. In this case each access to memory first requires a read from the pointer table, followed by access to the actual memory block. Accessing a pointer table adds extra memory operations, and this approach shifts the distribution problem from the dataset itself to this pointer table. A simpler and more predictable distribution would thus be the following:

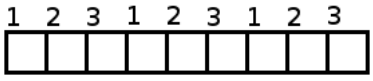

Figure 5.6: Implementation: Max-distributed global memory access pattern

In this example each thread simply allocates an amount of blocks equal to the maximum of all threads. This comes with an obvious downside: a lot of memory is wasted. Especially if the required amounts of memory greatly diverges between work-items this could result in a lot of wasted memory. Due to the distribution it might however lead to a higher performance of the entire program. In the example shown, memory blocks are accessed more linearly and more locally. With a prefix-sum this distance is bound by memObjCount(p(x)), which depending on the use-case could be any number. Figure 5.7 demonstrates that with a max-distribution the distance between two objects accessed in parallel is upper bound by $|p| - 2$.

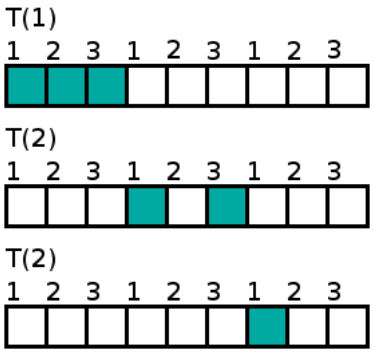

Figure 5.7: Implementation: Max-distributed global memory access pattern

Because the maximum number of blocks and the number of work-items are known, the work-items can also calculate the pointers to each block they should use without requiring a memory pointer table, thus without introducing extra memory read operations. Whether an application benefits from this increased locality depends on the use-case and the effectiveness of the memory controller and caches on the device, and can be observed by quantitative experiments.

Chapter 6

Performance and results

In this chapter we discuss the performance of KMA. We first analyse the theoretical performance, after which we present the results of a series of benchmarks to give a good insight in the performance of our heap allocator when used in real applications.

6.1 Theoretical performance

6.1.1 Complexity

Queue The memory allocator is based on a lock-free implementation of a queue that uses CAS to ensure the state is always correct. Like any queue, our implementation consists of two methods: enqueue and dequeue. To avoid the ABA problem, these methods store a pointer in an encoded format, as we explained in section 5. An *encoded pointer* consists of a *tag* and an encoded address we call *index*. A couple of helper functions are defined for recurring operations:

- *TAG()* and *IDX()* are bit-shift operations that for an encoded pointer return the tag value and the index respectively.
- *PTR()* returns an encoded pointer given a index and a tag.
- *ptr2idx()* is a function that converts a memory address to an index value.
- *idx2ptr()* converts an index value to a memory address.

On the next pages we present a complexity analysis of the *enqueue()* and *dequeue()* operations.

The enqueue algorithm is implemented as follows:

```
int enqueue(queue *q, queue_item *item) {
  queue_item *tail;
  uint32_t idx, tag, tailidx, nextidx;

  tag = TAG(item→next);
  item→next = PTR(0, tag-1);
  tag++;
  idx = ptr2idx(q, item);
  while(1) {                                                          O(∞)
    tailidx = q→tail;
    tail = idx2ptr(q, tailidx);
    nextidx = tail→next;
    if(q→tail == tailidx) {
      if(IDX(nextidx) == 0) {
        tag = TAG(nextidx);
        if(atom_cmpxchg(&tail→next, nextidx, PTR(idx, tag)) == nextidx)
          break;
      } else {
        tag = TAG(tailidx);
        atom_cmpxchg(&q→tail, tailidx, PTR(nextidx, tag));
      }
    }
  }
  tag = TAG(tailidx);
  atom_cmpxchg(&q→tail, tailidx, PTR(idx, tag));
  return true;
}
```

As one can see all steps within the while-loop execute in constant time. The while loop itself, though, is defined as an infinite loop. If one particular work-item is always interrupted after preparing its values but before committing its changes with the atomic CAS operation, it never ends. This effect is called starvation, and causes the work complexity of this algorithm to be infinite in theory. In practice though, every time that the atomic CAS operation fails for a work-item, it must have succeeded for another. This means that for every iteration of this algorithm by one work-item, there is at least one work-item that made progress. Although this does not mean that execution time of one work-item is always constant, every iteration in time results in one successful dequeue operation. The work complexity of this algorithm thus is $O(C * n) = O(n)$ where n is the number of work-items, and C a constant whose value depends on the scheduling algorithm used by the GPU. The lower bound work complexity for this algorithm is $\Omega(1)$.

The dequeue algorithm is implemented as follows:

```
queue_item *dequeue(queue *q) {
  queue_item *head;
  uint32_t tag, nextidx, tailidx, headidx;
  while(1) {                                                          O(∞)
    headidx = q→head;
    head = idx2ptr(q, headidx);
    tailidx = q→tail;
    nextidx = head→next;

    if(headidx == q→head) {
      if(IDX(headidx) == IDX(tailidx)) {
        if(IDX(nextidx) == 0) {
          return NULL;
        }
        tag = TAG(tailidx);
        atom_cmpxchg(&q→tail, tailidx, PTR(nextidx, tag));
      } else {
        tag = TAG(headidx);
        if(atom_cmpxchg(&q→head, headidx, PTR(nextidx, tag)) == headidx)
          break;
      }
    }
  }
  return head;
}
```

The complexity analysis for the dequeue operation is similar to that of the enqueue routine. Assuming starvation does not occur, the work complexity of this routine again is at worst $O(C * n) = O(t)$. This corresponds with sequential execution multiplied by a constant. The lower bound work complexity for this algorithm is $\Omega(1)$.

Allocator The in-kernel memory allocator relies on the queue mentioned earlier for managing the free-list of superblocks. The implementations of both *malloc()* and *free()* thus share the complexity with the queue implementation.

The first step of allocation is determining the right size class for the requested block size. This operation is a sequence of calculations that can be done in constant time.

The second step is to reserve a slot for a block within a superblock. If no superblock is attached to the right entry of the hashmap, the dequeue operation is called to obtain a new superblock. Otherwise the right superblock is taken from the hashmap. Reservation of a block within this superblock is an atomic exchange operation, which again might fail infinitely if work-items keep being pre-empted. Like the queue algorithm, there is no reason to assume this makes the algorithm run worse-than-sequential, although again the complexity of this reservation operation and thus of the second step is of $O(\infty)$.

The final step is to find which slot is still free and reserving that particular slot. The availability of these slots is stored in a bitmap at the end of a superblock. Using an atomic-or operation, a single bit can be set. Judging by the return value of this operation, it can be verified if it was already reserved before executing the atomic-or operation or not. Again this procedure ideally just takes constant time, but worst case it has to be repeated infinitely if the block it tries to allocate is taken before the atomic operation succeeds. The complexity of this operation is again upper bound by $O(\infty)$.

The algorithm used by the free routine is not very complex apart from the dequeue operation. A normal free is nothing more than atomically setting the bit in the bitmap to 0, and then atomically increasing the slot number by one. The exact opposite happens of allocation, and with the same complexity.

Although the theoretical complexity does not promise very good performance, the memory allocator does not perform that poorly in practice. In the worst case the lock-free allocation algorithm will obtain the performance of sequential execution, which is expected in a globally shared data structure and not worse than the expected performance of any locking algorithm. To better understand the behaviour and performance of KMA in real applications, we will rely on the results of a series of benchmarks. If these benchmarks complete execution, the upper bound complexity is never reached.

6.1.2 Memory overhead

To comment on the amount of memory that is lost due to fragmentation, we must first make a distinction between static- and dynamic memory overhead. Static memory overhead is the memory lost on static data structures. Depending on the size and number of allocated blocks there also is a surplus of memory wasted, which we attribute to dynamic memory overhead.

Static memory overhead To keep track of the state of the entire heap, three datastructures are used: a queue for the free-list, a hashmap for the active superblocks and a bitmap for the available blocks inside a superblock. Each of these structures take some space that can not be used for user data. The hashmap consists of one pointer for each size-class. The number of size classes in our implementation is equal to $({}^2log(sb) - 1)$, where sb is the size of a superblock. The free-list is a combination of two encoded pointers, each requiring four bytes, and two regular pointers. To store the hashmap and the free-list in the state of a heap with 4KB superblocks, along with the total size of the heap, 64 bytes are reserved in total at the beginning of the heap on 32-bit platforms and 120 bytes on 64-bit platforms.

In addition, as explained in section 5, the queue algorithm forbids the queue to run completely empty, wasting one full superblock of memory.

Also static is the header for each superblock. It consists of a next-pointer for the queue, the size of the blocks inside the superblock and a 32-bit volatile int2 value holding the number of free slots and the number of total slots. In total, each superblock will thus have a header of 12 bytes on 32-bit and 16 bytes on 64-bit platforms.

If we put all these numbers together, the total static overhead o, with n superblocks of size sb, can be calculated for 32-bit platforms with:

$$o = sb + (4 * ({}^2log(sb) - 1)) + 20 + (12 * n) \tag{6.1}$$

For 64-bit platforms the overhead o can be calculated with:

$$o = sb + (8 * ({}^2log(sb) - 1)) + 32 + (16 * n) \tag{6.2}$$

Dynamic memory overhead The memory overhead within a superblocks data structure consists of three categories: the bitmap used to keep track of free blocks, the amount of bytes sacrificed for non-optimal block sizes and the amount of memory of blocks not allocated or wasted due to rounding errors.

The number of bytes in this bitmap can be calculated by rounding up the number of elements inside a superblock to the next multiple of 32, and dividing this number by 8. For blocks of a maximum size this will be 4 bytes, for blocks of the minimum size of 4 bytes this could be as much as 128 bytes.

The upper bound on the amount of memory wasted on fragmentation caused by a non-optimal block size cannot be calculated deterministically for all cases. The heuristic we use to determine a block-size is explained in section 5.1.1. This memory wasteage for a small block-size is no more

than the size of one block for each superblock, and for a bigger block-size this wasteage is no more than three bytes multiplied with the number of blocks inside the superblock.

The last type of memory overhead is caused by underutilised superblocks. In theory it could occur that a program allocates full superblocks of 4-byte blocks, making the superblocks inactive, and then frees all blocks but one per superblock. These freed blocks cannot be re-allocated, because the superblocks are in an inactive state. This leads to a total waste of memory of over $\frac{4092}{4096}$. Re-attaching pages that have been previously occupied is one strategy to reduce this memory wastage, but at the price of performance.

6.2 Benchmarks

For a better understanding of how this allocator performs in real application, several benchmarks are designed and tested on a variety of hardware, most of which has been made available on the DAS-4 supercomputer[1]. The available hardware and its configuration is listed in table 6.1. Besides running our benchmarks on a variety of GPUs, we also executed the test programs on several CPUs to be able to show the portability of this code. Additionally, both AMD HD6850 and AMD HD7970 graphics cards were available for tests, but due to problems explained in section 2 the memory allocator did not function properly on the AMD GPU platform. Unfortunately, there were no OpenCL compatible ARM-based devices available for testing.

Device	Type	Cores	Clk(MHz)	(V)RAM	Bitness	Software
NVIDIA GeForce GT640	GPU	384	901	2GB	32	Cuda 5.0.35
NVIDIA GeForce GTX480	GPU	448	1215	1.25GB	32	Cuda 5.0.35
NVIDIA GeForce GTX680	GPU	1536	1006	2GB	32	Cuda 5.0.35
NVIDIA Tesla C2050	GPU	448	1150	3GB	32	Cuda 5.0.35
NVIDIA Tesla K20m	GPU	2496	706	5GB	32	Cuda 5.0.35
AMD FX-6300	CPU	6	3500	8GB	32/64	AMD APP 2.8
Intel Xeon E5-2620	CPU	6(12)	2000	24GB	32/64	AMD APP 2.8
Intel Xeon E5620	CPU	4(8)	2400	24GB	32/64	AMD APP 2.8
Intel Xeon X5650	CPU	6(12)	2670	24GB	32/64	AMD APP 2.8

Table 6.1: Hardware available for experiments

6.2.1 "Low-level": Memory allocator

Allocation time To get an idea of the time it takes to allocate and free a block of memory, we execute a test kernel that repetitively calls *malloc()* and *free()* in a tight loop. In this kernel a single work-item repetitively allocates and frees a block of memory. We expect that the execution time of this benchmark grows linearly with the number of allocations, and that an approximation of the resulting curve can be given with the linear formula $t = ai+b$. If our assumption of linearity holds, we can find both the linear factor a and the constant overhead b for this linear equation. Specifically, we are interested in finding a, the sequential time per iteration.

Results of this experiment can be found in figure 6.1. The execution time in seconds is measured from the moment the kernel is enqueued on the command queue to the moment *clFinish()* returns.

[1] http://www.cs.vu.nl/das4/

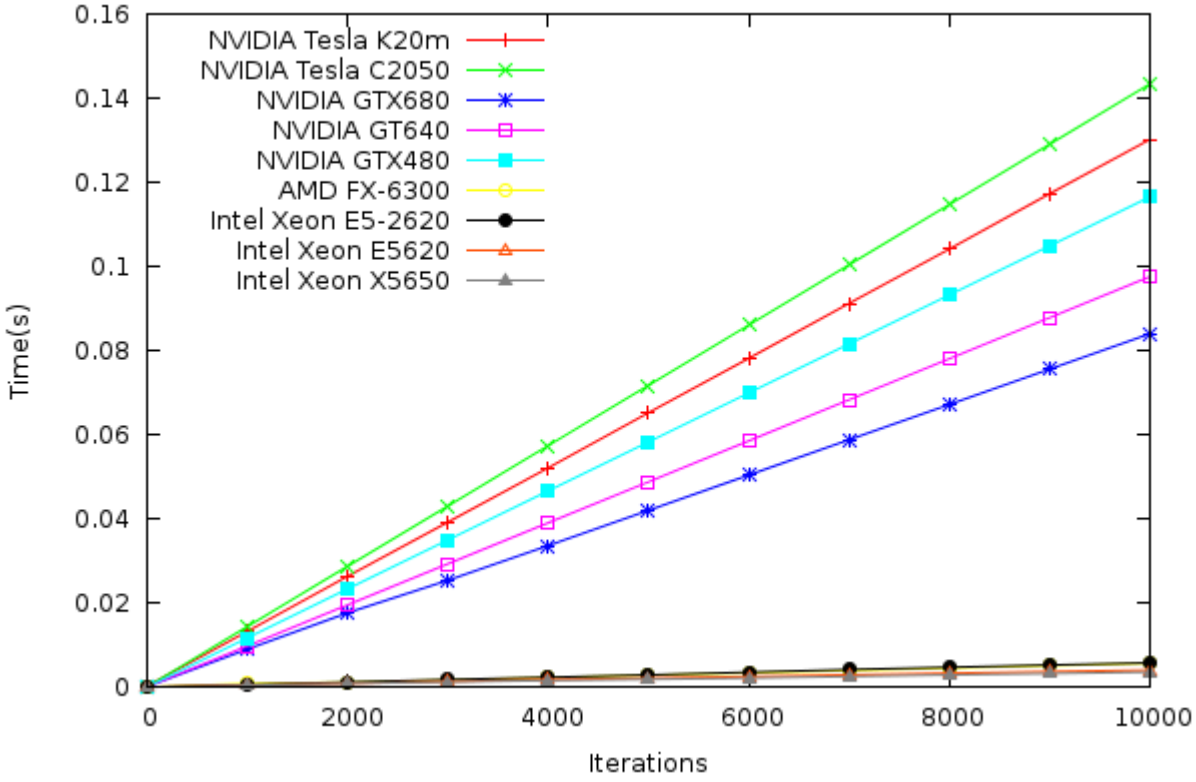

Figure 6.1: Single-work-item performance of the low-level memory allocator

As can be observed there is indeed a linear relation between the number of calls to *malloc* and *free*, and the execution time. Because the curve seems to originate from the point $(0, 0)$, we can neglect the overhead introduced by launching a kernel. The time per iteration a can thus be approximated simply by dividing any of the results with the number of iterations. The resulting time on an NVIDIA GT640 is approximately 9.8ms. CPUs finish this benchmark much faster, with an average of $0.6\mu s$ per allocation for the AMD FX-6300. For comparison, the authors of the Xmalloc paper[23] reports a latency of $166\mu s$ for the Cuda *malloc()* operation, and $50\mu s$ for its XMalloc routine on an NVIDIA Tesla C1060, an older but high end GPU based on NVIDIAs GT200 series of GPUs.

To analyse the error in this approximation we could consider a run with very few iterations. For two iterations on the NVIDIA GeForce GT640, we measured an average execution time of 0.000067s. Samples varied between 0.000065 and 0.000069, a difference of merely 6%. The approximated time per iteration is $\frac{0.000067}{2} \approx 3.33 * 10^{-5}$. The value of the constant b can be approximated by $0.000067 - (2 * 0.98 * 10^{-5}) \approx 0.47 * 10^{-4}$, which is 0.05% of the total execution time for 10000 iterations. Distributed over these 10000 iterations, the absolute error in the time per iteration calculated earlier is in fact insignificant.

A quick glance at the numbers for the AMD FX-6300, 0.000039s for 2 iterations and 0.005694s for 10000, learns that the error here can not be more than 10 times larger. Even with an overhead of 0.5% we expect the measuring error to be of larger influence on the time per iteration obtained than the approximated set-up time.

Scalability To test the scalability of the memory allocator, we executed the same test program with different parameters. For this benchmark, the iteration count is always 100. Within this loop, a small amount of memory is allocated, the exact amount of memory differing between work-items.

This benchmark is repeated with a various number of work-items, each work-item thus performing 100 allocations and frees.

The benchmark is executed with three different allocation schemes. In the "no variance" experiment, all work-items allocate 4 bytes of memory. In the "low variance" experiment, half of the work-items allocate 4 bytes of memory and the other half allocates 8 bytes. To break patterns, a single work-item also allocated 4000 bytes of memory for every iteration. In the "high variance" experiment, each work-item allocates 2^n bytes of memory for $0 \leq n < 5$. The value of n depends on the work-items ID and the iteration count, increasing the variance in allocated memory. Again, a single work-item also allocates 4000 bytes of memory in every iteration. For all three experiments we hope to see at worst a linear relation between execution time and the number of work-items. A linear curve will mean that parallelism is fully utilised even for a small number of work-items and that work-items do not counteract progress of other work-items. Figure 6.2 shows the results of these experiments.

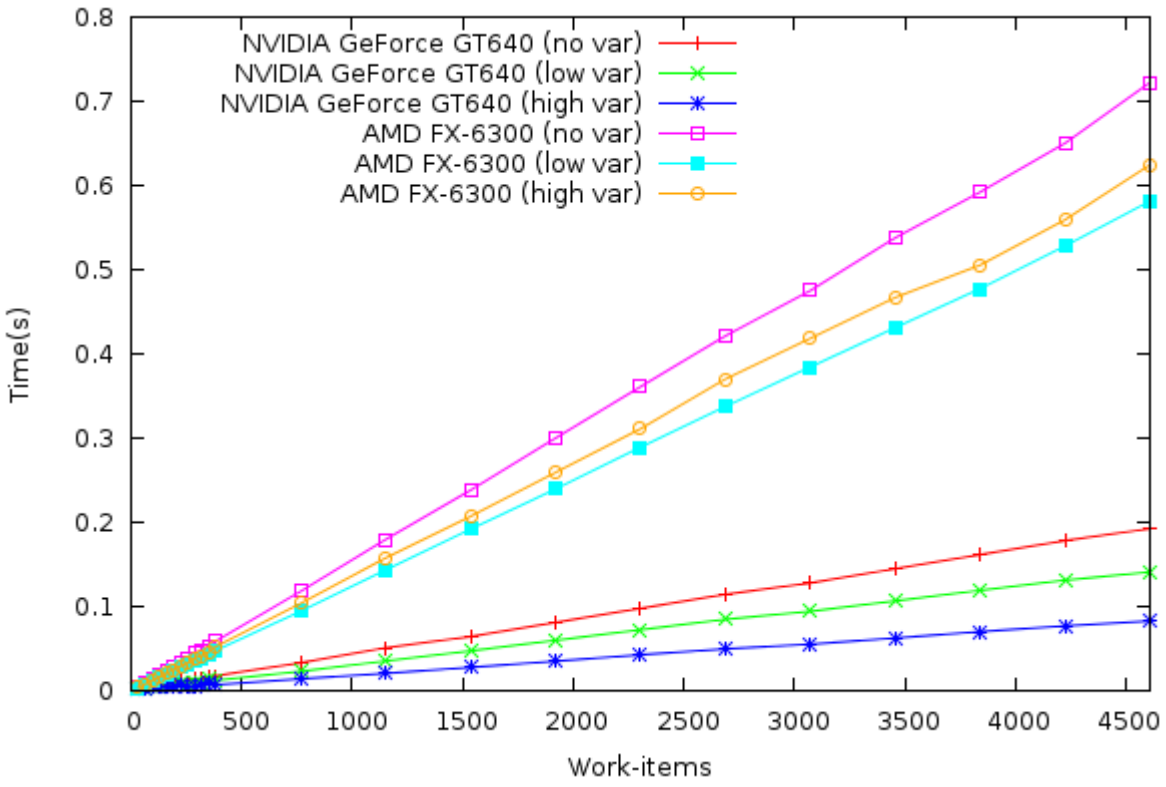

Figure 6.2: Parallel performance of the low-level memory allocator

A few conclusions can be drawn from this graph. First of all, the execution time of the program grows linearly with the number of work-items and thus with the total number of calls to the memory allocator. The fact that there are more work-items to handle these cases makes no difference with KMAs lock-free algorithms. We expected that the memory allocator would not perform worse than sequential execution, and in absolute numbers this memory allocator does not. The nearly straight line seems to confirm that the algorithm does scale linearly in this case.

If the time found for 4608 work-items on the NVIDIA GT640 GPU is divided by the number of allocations, the time per single allocation and deallocation pair is estimated around $\frac{0.192761}{460800} \approx 0.42\mu s$, a major difference from the earlier found $9.8\mu s$. The estimation based on 4608 work-items is closer to the actual time the GPU will take for the allocation and freeing because when there are more work-items, the GPU can schedule a different work-item while waiting for the memory

operations of the first to complete. This mechanism greatly increases the performance of the overall system even if there were just one core available.

Our second observation is that by increasing the variance of allocated memory block sizes, the efficiency of this memory allocator increases. This is to be expected when analysing the algorithm, because there are more active superblocks and each is accessed by less work-items. As these structures are not dependent on each other, more parallelism can be achieved.

This property is not of much use for the use-cases we have identified, where work-items tend to allocate blocks of equal size to create irregular data structures. However, with the ArrayList implementation this property could lead to higher performance, as the total number of allocated memory could vary depending on the amount of required objects for the work-group.

On the AMD FX-6300 GPU we see different results. Although in the previous experiment the CPU perform a lot better than the GPU due to a higher sequential performance, it is unable to keep up in its massive-parallel execution. Execution of these experiments on the CPU was approximately three times slower than on the GPU. Moreover, in cases where the GPU was able to utilise parallelism better by adding more variation in the block size of allocated blocks, the CPU already fully utilises parallelism when there is low variation. More variation in block size just results in more work being done, which translates to higher execution times.

6.2.2 "High-level": ArrayList

Ideally, to evaluate the performance of the ArrayList, a similar experiment should be conducted as in section 6.2.1. Unfortunately, as explained in section 2.4, there is no practical and performant way to do global synchronisation. Without synchronisation, it is impossible to decide when all elements should be freed while maintaining a predictable state of the global data structure. When using global sync by cutting up the kernel, the run-time will be dominated by the cost of synchronisation,

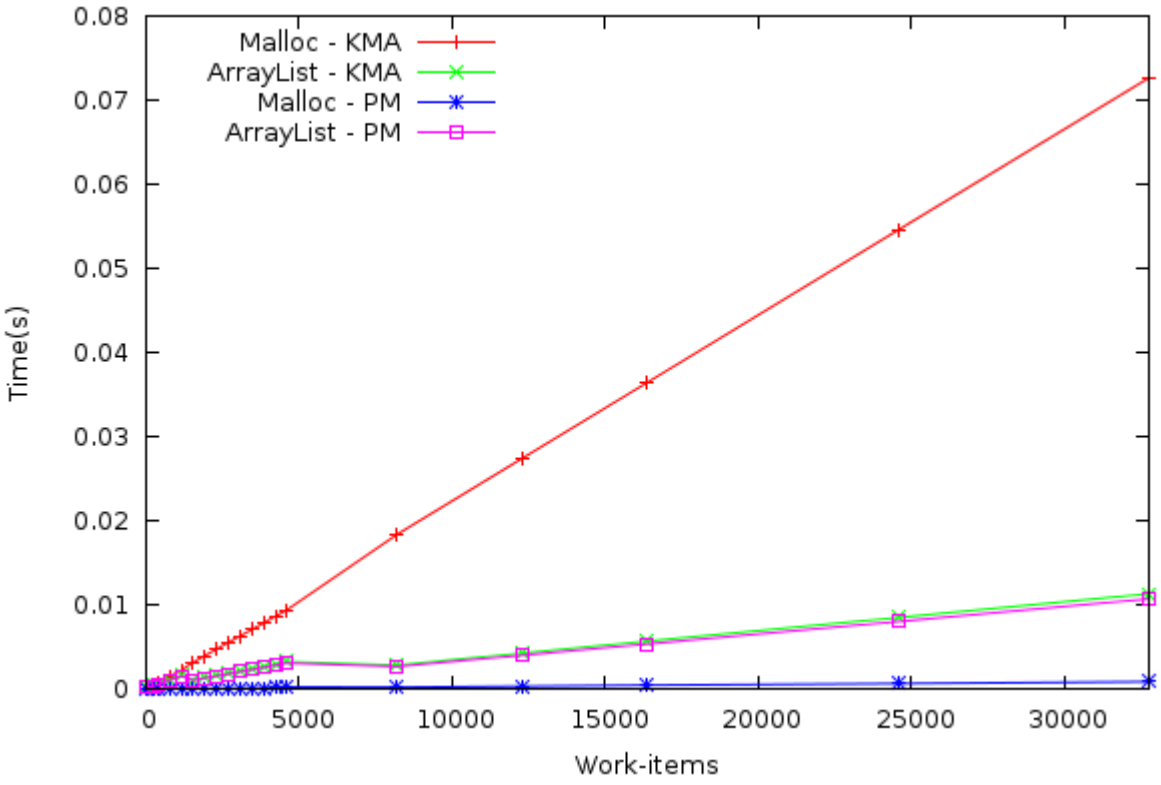

Figure 6.3: Comparing ArrayList to Malloc - NVIDIA GeForce GT640

and thus will not show the performance of the routine itself. This makes it impossible to use the same experiment to compare performance figures with those of the low-level allocator experiments conducted earlier.

Instead, we focus on the performance of just the *malloc()* routine, comparing the low-level heap allocator (*malloc*) with the high-level ArrayList object. We set up a simple test case where each work item allocates either 4 or 8 bytes of memory 10 times in a row. In the malloc case, every thread allocates its own memory. In the ArrayList case, the *clArraylist_grow_local* routine is called. Allocated memory is not freed, so from the programmers point of view the two programs have the same end result. This test is conducted with both the KMA-heap and the poormans-heap as a back-end. We measure the execution time for a varying number of threads, from the moment the kernel is enqueued until clFinish() returns. We expect the ArrayList to perform better than the low-level memory allocator in absolute numbers. Eventually we expect the execution time to scale linearly with the number of work-items because when the maximum work-group size is reached, adding extra work-items will only linearly increase the amount of work. Results on the NVIDIA GeForce GT640 can be found in Figure 6.3.

The graph shows a clear benefit for using the ArrayList implementation over KMAs low level memory allocator like we expected. For a very small number of threads the execution time is negligible, but when a few hundred work-items or more are used the performance benefit is clear. The outlier at 8192 work-items is likely caused by non-ideal work-group sizes for previous thread configurations. For 16384 threads on a GPU the ArrayList testcase executes 7 times faster than the Malloc testcase. Since both curves appear to be linear, apart from the outlier, the absolute difference in time between the two will only grow further.

What is also shown is that the poormans allocator without the ArrayList is by far the fastest solution, outperforming KMA by a factor of 72 with 32768 threads and continuing to widen this gap. Using just the poormans heap even outperforms the ArrayList with the Poormans-heap.

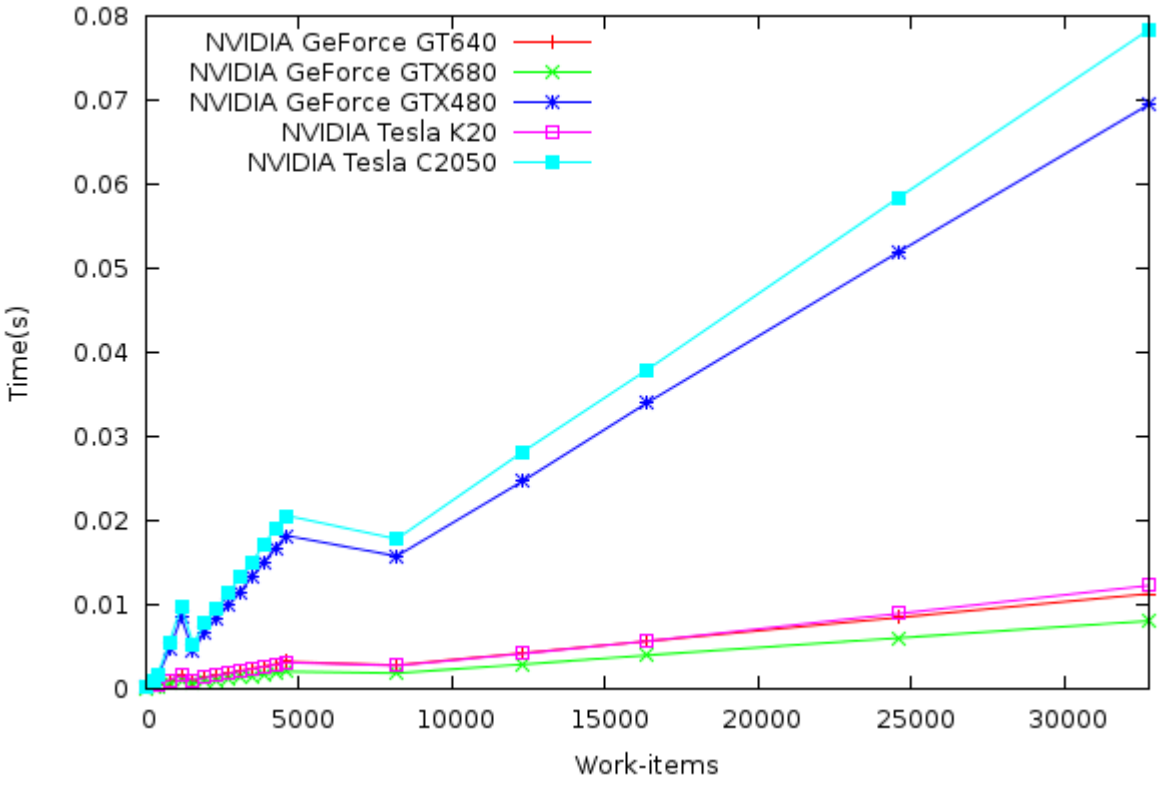

Figure 6.4: Peformance of ArrayList on different GPUs

Finally, when we compare the performance of the ArrayList between the two different back-ends (KMA and PM) we see a nearly identical performance. The prefix-sum algorithm dominates the execution time of the executed benchmark, while the performance of the memory allocation back-end has become a negligible factor. We expect to see a similar effect in real-life applications, where the execution time of a program is not dominated by the performance of the memory allocator, but rather by the algorithm it implements.

To show the portability of the ArrayLists performance with KMA back-end, we repeated this experiment on a series of graphics cards available in the DAS-4 supercomputer. The results are shown in graph 6.4.

Clearly its performance is roughly equal across all Kepler-based NVIDIA graphics cards (GTX680, GT640, K20), GTX680 being the fastest. On the previous generation Fermi cards the test program executes roughly 9 times slower, and 4 times slower when not using the ArrayList. This is explained by the increase in number of cores and clock-speeds as can be observed in table 6.1, along with the increased memory bandwidth, and presumably the dual-issue logic and other architectural improvements in the newer generation cards[35].

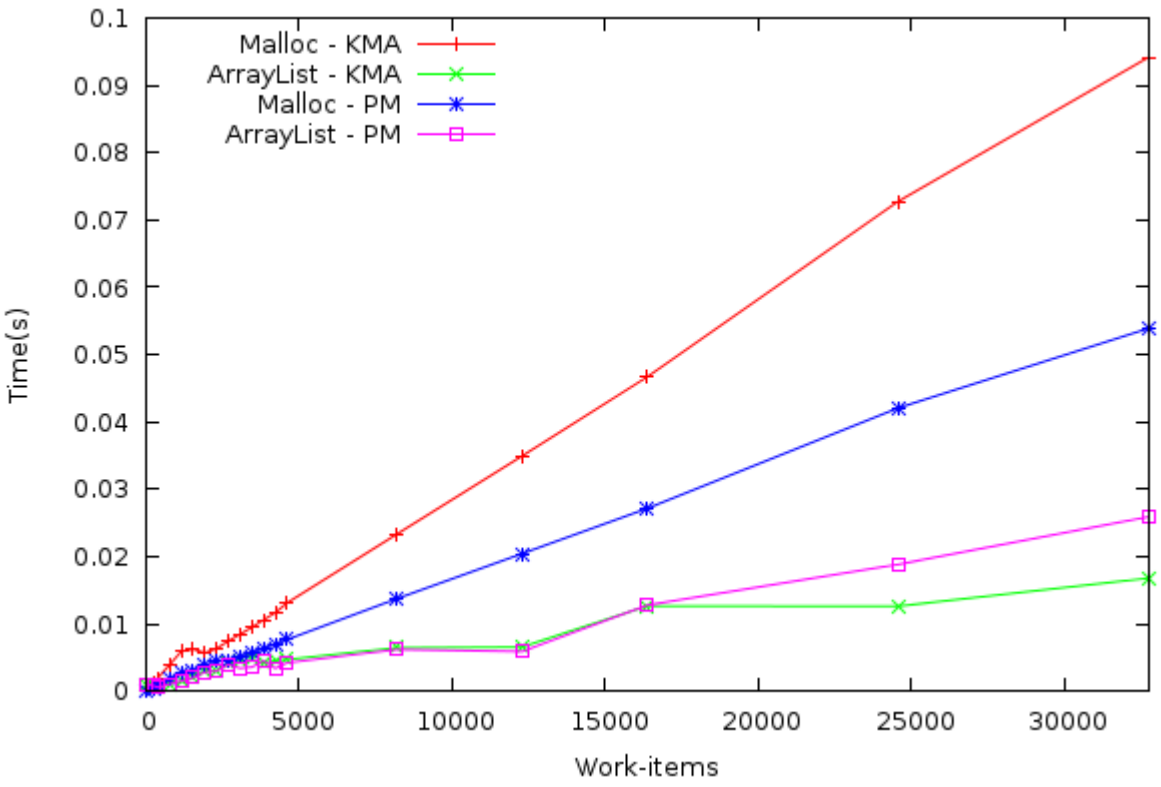

Figure 6.5: Comparing ArrayList to Malloc - AMD FX-6300

Figure 6.5 shows the same experiments ran on the AMD FX-6300 CPU using AMD APP. Again the ArrayList shows a performance improvement over just using the KMA-1 allocator, albeit a smaller difference because even the prefix-sum reduction now only has six cores to execute on, even though the algorithm allows for greater parallelism.

Also similar is the fact that the Poormans heap here is faster than KMA, although the difference is a lot smaller. On the CPU though, using the ArrayList is always quicker than calling the low-level *malloc()* routine separately. Most likely the atomic operations required for a memory allocator, no matter how complex, are a lot more expensive on the CPU than on an NVIDIA GPU.

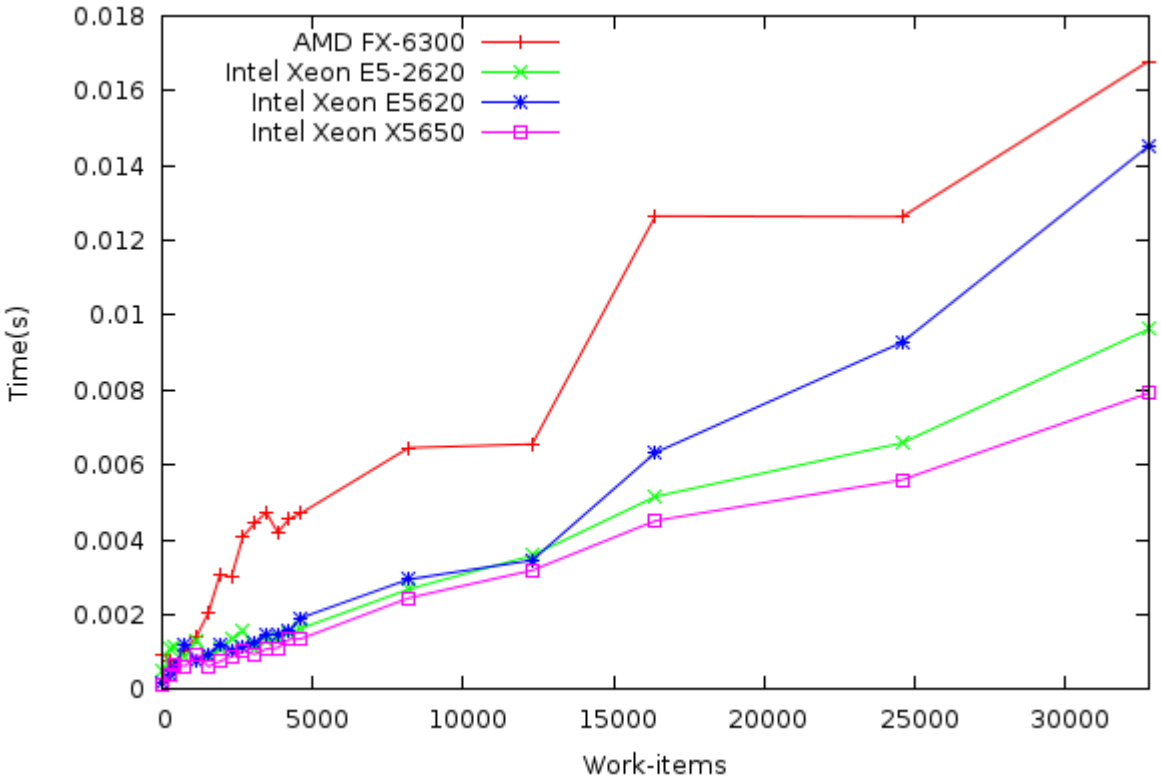

Figure 6.6: Peformance of ArrayList on different CPUs

Figure 6.6 shows the performance of the ArrayList for different CPUs. The performance seems to roughly correspond to the performance ratings given to these cores by the CPU manufacturers, even though these are often based on floating point calculations whereas our test program uses integers. The ArrayList benchmark seems to benefit more from extra cores than raw clock speed, as we observe from the minor difference between the Intel Xeon E5-2620 and the higher-clocked (but older) Intel Xeon X5650.

Although the usability of the ArrayList depends greatly on the use-case, with limitations like expensive global synchronisation if the data must be free'd or otherwise be stable, the experiment does show that the "two-layered" KMA is not only a useful tool for meeting a programmers dynamic allocation needs, but also offers the right level of support for optimisations if the use-case permits.

6.2.3 Use-case: Tree construction

To show the usability of KMA we implemented a practical use-case. This test program is designed to convert a directed graph represented by a list of arbitrary edges into a binary search tree of nodes with its edges represented as pointers to different nodes. One can imagine algorithms like graph-colouring or shortest path working with this graph.

In the binary search tree, a node object contains a key, left and right pointers, and a singly linked-list to store the edges of the graph. An edge is an object containing a pointer to the next object in the SLL, and a pointer to the destination node. The node and edge objects are created by using the low-level KMA-1 malloc routine, after which they are attached to the searchable binary tree. If one attempts to add a node that was previously added, the node will be free'd again.

The algorithm used to store this directed graph is a simple lock-free binary search tree algorithm that allows to attach nodes and search for their existence based on the key. It does not need to

know how these are allocated. The add method is implemented as following:

```
bool clTree_add(*tree, *node) {
    uintptr_t *cur = &(tree->root);
    clTree_node *cn;
    node->left = node->right = NULL;
    while(1) {
        cn = atom_cmpxchg(cur, NULL, node);
        if(cn == NULL) return true;
        if(cn->key == node->key) return false;

        if(node->key < cn->key) cur = &cn->left;
        else cur = &cn->right;
    }
}
```

As input, the test program gets only a list of edges represented as {*source_id, sink_id*} tuples, and the number of edges available in the list. As the node ids and number of links per node are not known in advance, it is difficult to use static sized data structures like arrays or hashmaps. By using a dynamic memory allocator instead, we minimise memory overhead while keeping code easy to read.

Usability Using a heap starts with compiling the required heap code by adding its source files to the program. The exact code used to compile the two source files in the host program, clSBMalloc.cl and clIndexedQueue.cl, depends on the programming language the host program is written in, but does not differ from the code used to compile the actual kernel.

The next step is to create a heap on the host system by calling the clSBMalloc_create() routine described in section 4.2.1. After this, the heap is ready to use by any kernel that receives the returned cl_mem object as a parameter.

The source code for adding a node to the tree shows the use of our memory allocator within this program. The following routine finds a node in the tree, and adds it if it does not already exist. Some keywords, like "__global" and "struct", and typecasts are omitted for readability.

```
graph_node *ensure(*heap, *tree, int key) {
    graph_node *node = NULL;
    while(node == NULL) {
        node = clTree_get(tree, key);
        if(!node) {
            node = malloc(heap,sizeof(graph_node));
            if(!node)
                continue; /* or fail */
            node->tree.key = key;
            sll_init(&node->links);
            if(!clTree_add(tree, &node->tree)) {
                free(heap, node);
                node = NULL;
            }
        }
    }
    return node;
}
```

The code for adding a link to a node, where *item* is the currently processed edge, then looks as follows:

```
source = ensure(heap, tree, item->source);
sink = ensure(heap, tree, item->sink);
link = malloc(heap, sizeof(graph_link));
link->sink = sink;
sll_add(&source->links, &link->q);
```

The code examples we have shown in this paragraph should not be fundamentally different from code written for a "regular" application that would run on the host system. The allocation and freeing of blocks are handled by the low-level APIs of KMA, like they would be handled by libc for regular C programs.

Performance For understanding the performance of KMA in a real life application, we used these tree routines to create a tree from a directed graph containing 65536 edges and up to 10000 nodes. This test case was executed both on the CPU and GPU with a varying number of threads. To show the impact of a full memory allocator, performance is compared to the Poormans-heap. To contain the memory requirements, the Poormans-heap test code is slightly less naive than the KMA code: The Poormans-heap code holds on to an allocated object for a possible next iteration while in the KMA code this block would be freed and re-allocated on the next iteration. For a fair comparison between the non-caching KMA code and the caching Poormans-heap code, we also used this caching technique with KMA.

The static set-up time of the data structures of this test program on the GPU when KMA is initialised with 512 superblocks of 4KB each is 1.5ms. For the "poormans"-heap the total set-up time is 0.2ms. This means approximately 1.3ms is spent on the single threaded initialisation kernel that enqueues all the superblocks to the free-list. On the CPU similar numbers are observed: 1.15ms with KMA versus 0.1ms for the poormans-heap.

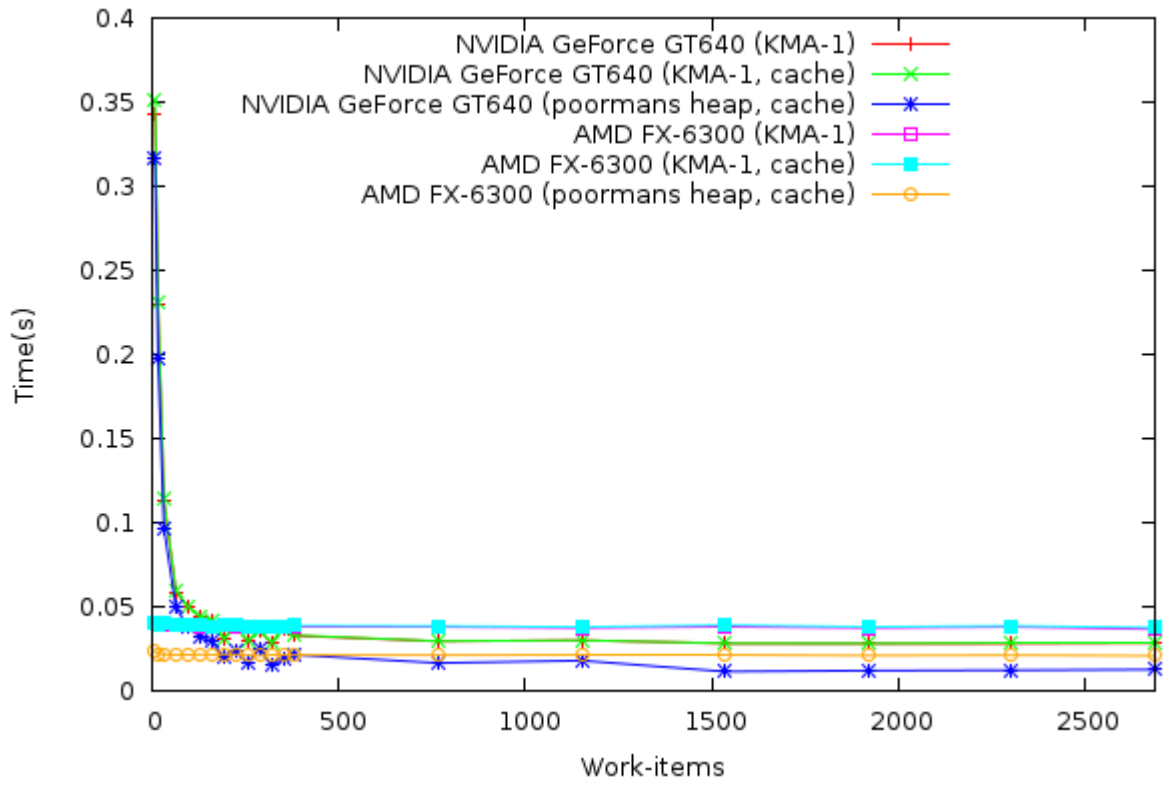

Figure 6.7: clTree performance: KMA-1 vs. poormans heap

The execution time relative to the number of threads is shown in figure 6.7. The first thing to observe is that after about 400 threads, performance does not increase any further on the GPU, apart from the small jump at 1536 threads. The latter can probably be explained by executing with a more favourable work-group size, but all in all the difference is minor.

The difference between KMA with and without caching is negligible. The amount of superfluous allocations done is limited by the checks, and the code required for this caching does not outperform the memory allocator itself. This means that the difference in performance between KMA and the poormans heap is caused entirely by the allocation algorithm.

In absolute numbers, this test program executes in approximately 28.6ms when using the KMA back-end, where the program finishes in 12.6ms when using the Poormans-heap back-end. With the naive usage of KMA the total overhead is 56% of the execution time. On the AMD FX-6300 CPU, performance does not change with the number of work-items. This is not surprising as adding more work-items does not increase the amount of work, while parallelism is already exploited with very few work-items because there are only six cores available. Here we observe times of 37.9ms and 20.7ms respectively, meaning KMA-1 adds a performance overhead of approximately 45%.

At first this seems like a big price to pay for the full functionality of a memory allocator, and in some cases it might be. However, most use-cases do not end after constructing this tree, but rather this is the first step in a bigger algorithm. The relative overhead for using KMA in a complete algorithm is likely a lot smaller in these cases, while the use of KMA does lead to a better code readability, more flexibility and possibly better memory usage. If the use-case permits, programs can also benefit from the prefix-sum reduction based allocation algorithm tested in 6.2.2, further reducing the overhead.

Portability To show the portability of the clTree program, performance was measured on the available NVIDIA GPUs in DAS-4. We expect some absolute differences in execution time between

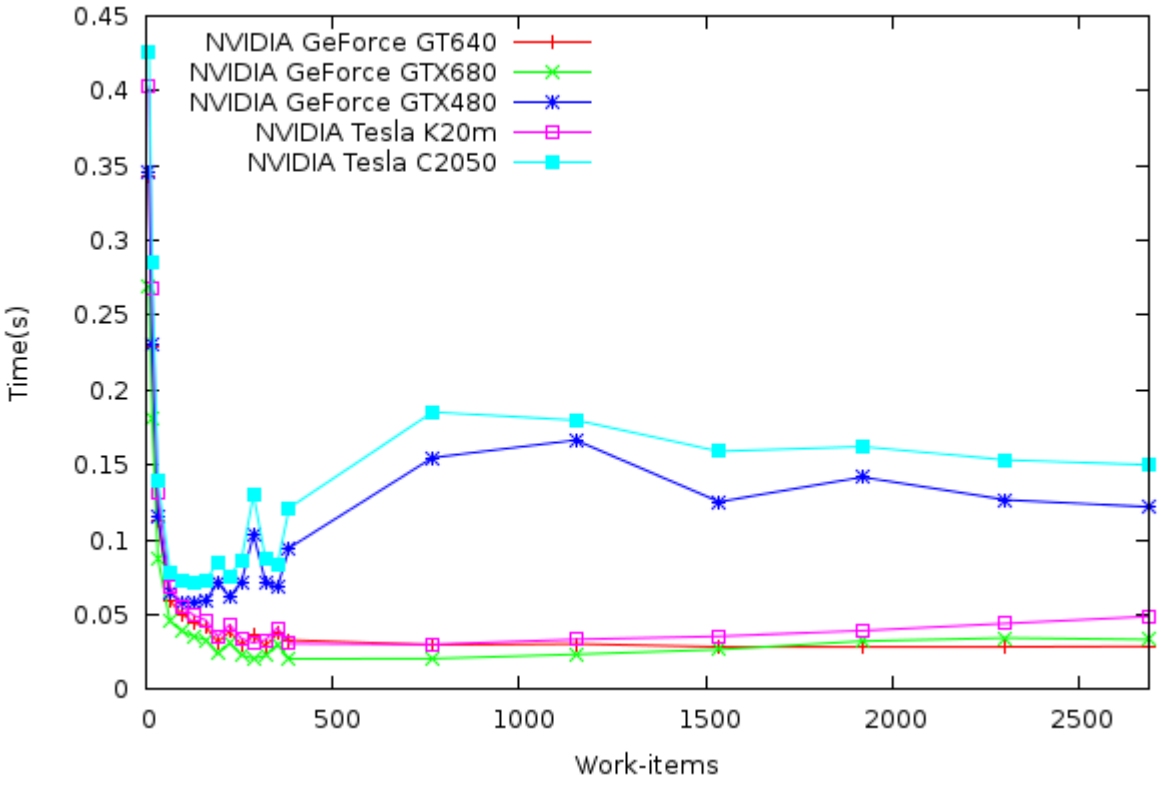

Figure 6.8: clTree performance on different GPUs

the different GPUs. The shape of the graph, however, is expected to be similar to those found for the GT640 in the previous paragraph. Our results are shown in figure 6.8.

The trends in this graph for the Kepler cards are as expected, where little difference is found between these cards. The "sweet-spot" for these cards using this particular dataset is around 768 work-items. Adding more work-items will only increase the pressure on the malloc algorithm, that does not permit for more parallelism. As a result, the overhead increases with the number of work-items, as can be seen clearly on the K20m.

Judging by the dip in the graph, on the Fermi-based cards parallelism seems utilised entirely when launching approximately 128 work-items. With its more complex schedulers compared to Kepler-based GPUs[35], the overhead for adding more work-items on Fermi-based GPUs is larger, resulting in a greater reduction in performance for more than 128 work-items.

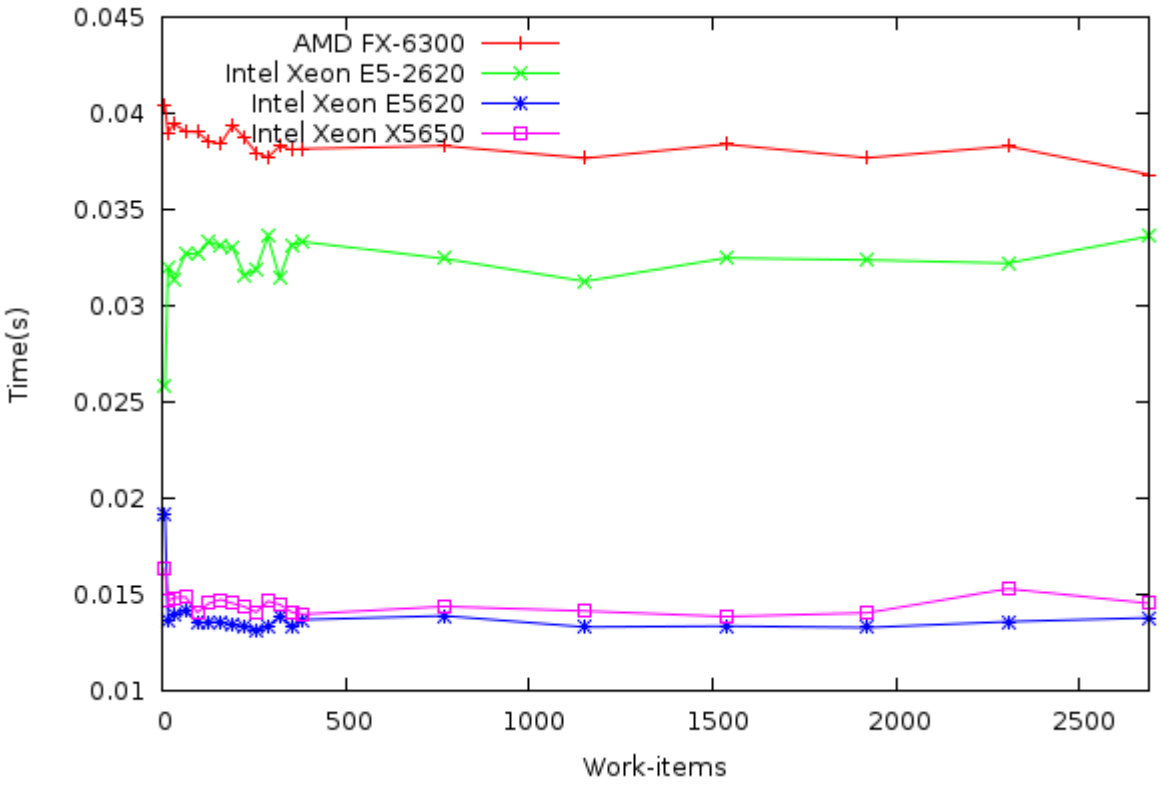

Figure 6.9: clTree performance on different CPUs

For CPUs, the performance differences are also small, although relatively the Intel Xeon E5620 and X5650 are up to three times as fast as AMDs FX-6300 desktop processor, as can be observed in figure 6.9. All trends are roughly horizontal, as we expected from a test setup where the number of work-items is always higher than the number of cores.

The difference in performance between the CPU and the GPU is small in absolute numbers. This is caused by the fact that the creation of a global data structure, like a tree, is not an efficient task to perform in parallel. a CPU has the benefit of a higher sequential performance. Our results show that a fast CPU can build up this tree twice as fast as a GPU. However, contrary to the creation of a tree, usage of such a structure can safely be done in parallel. When considering a program that both creates and the uses this tree, execution on a GPU might still offer a big performance benefit over a CPU.

Chapter 7

Discussion and further research

In this chapter we discuss our findings from the process of designing, implementing and testing a memory allocator. We evaluate the current KMA design and its use to OpenCL programmers, propose several areas where KMA could improve in the future and discuss some possible solutions to problems related to the portability of OpenCL across different platforms.

7.1 Trade-offs for KMA

As shown in section 4, algorithms that involve irregular data structures demand more flexibility than currently offered by OpenCL's restrictive, static host-side memory allocation. As a result, OpenCL is perceived as unsuitable for algorithms using these structures, even though there is no understanding of whether this is merely a software limitation or a consequence of hardware design.

To answer the question "Is OpenCL an efficient platform for irregular data structures?", we need to enable developers to experiment with these structures and draw conclusions from their work. Tools like a memory allocator will aid developers in doing just this. And if the answer to this question turns out to be negative, there are still algorithms that use these structures outside the critical path, that could benefit from an easy-to use heap regardless of its performance. This motivation led us to develop KMA.

Section 6.2.3 contains several code examples that show how the use of malloc() leads to simple and understandable code. Essentially, we argue that the complexity of dealing with dynamic memory management has been completely hidden in KMA behind the `malloc()` and `free()` calls. However, in practice, there are still a few issues that need to be addressed before this statement becomes 100% true.

The most severe problems we have encountered are related to the portability requirements amongst different OpenCL implementations. Code that ran correctly on NVIDIA GPUs with NVIDIA's compiler, and on AMD and Intel CPUs with AMD's compiler turned out to be impossible to compile using Intel's OpenCL platform and failed to properly execute on AMD's GPUs (as tested on AMD HD7970 and AMD HD6850). On the Intel platform, the lack of 64-bit atomic operations in combination with its 64-bit pointers makes it very tough to create a portable memory allocator. For the AMD GPUs, the problem is related to the way AMD's OpenCL interprets and implements the relaxed memory consistency model of OpenCL, as explained in section 2.4.

In terms of performance, KMA-1 adds a certain performance overhead by design (see Section 4). We did show ways to reduce this overhead in section 6.2.2, but in the end using KMA is yet another design decision any developer needs to take: the trade-off between the high-level KMA functionality, leading to higher productivity and maintainability, and its impact on performance.

7.2 Proposals for "KMA-2"

A lot of the design decisions taken for KMA are enforced by the current platform limitations discussed in Section 4. Improvements for KMA-2, both on performance and in portability, thus depend on improvements on the platform.

For *performance*, the main priority is to limit the memory wastage of KMA without sacrificing speed. For example, bigger blocks could be allocated when free blocks can be coalesced. This feature requires an algorithm based on a lock-free ordered linked-list instead of the less complex queue-based algorithm KMA uses for its free-list. Other areas where such complex algorithms could help is to reduce fragmentation of half-empty superblocks, requiring more advanced lock-free linked list algorithms that allow real unlinking of arbitrary nodes. Internal fragmentation can be reduced with more fine-grained control of the size of the allocated blocks within their superblocks. This last point might be achievable with current algorithms.

Literature has discussed several problems that need to be solved before lock-free algorithms for complex global data structures can be implemented. One big problem is the use of local cursors towards blocks as explained in section 4. Other problems that might arise with linked-lists are solvable by the use of CAS2[20], an atomic operation performing compares-and-exchanges on two disjoint memory locations. However, CAS2 is unlikely to be found in a modern computer architecture. Possibly the most viable solution right now to solve fragmentation would be the ability to use mutex locks in OpenCL to protect data instead of relying on limited lock-free algorithms. However, the performance impact of mutex locks might outweigh the memory benefit. Also current OpenCL implementations do not allow safe use of mutex locks, as we discussed in section 2.4, and the specification provides no suitable mechanism to implement one.

In section 5.3.2, we have sketched some interesting experiments that may improve the high-level data structure: using a different reduction algorithm might improve data locality, and global synchronisation could reduce the number of calls to the memory allocator. The latter experiment seems unlikely to give any interesting speed improvement. Comparison between the "Poormans-heap" and the KMA-1 heap back-end showed very little difference in performance, showing that the time spent on actual allocation is already relatively small with local synchronisation. Compared with the presumably large overhead for global synchronisation and the increased time spent on the larger prefix-sum reductions, the cost will most likely outweigh the benefit.

Using different reduction algorithms could be a more likely win in terms of performance. One of the downsides of the prefix-sum reduction is that the objects accessed in parallel will almost always be apart in memory by an (irregular) stride. This potentially limits the bandwidth available for memory operations, degrading performance. Being able to distribute the memory in such a way that adjacent objects will be accessed in parallel without adding new memory accesses will lead to higher bandwidth utilisation, in turn decreasing the run-time of any kernel using this high-level data structure.

For *portability*, we need to ensure that KMA-2 is functionally portable on all OpenCL platforms. From the perspective of KMA, this can currently only be solved by adding constraints to the design. It might be possible to eliminate the need of 64-bit atomic operations partially by encoding memory addresses as a 32-bit offset from a base pointer. A similar approach was successfully applied to encode the address of a superblock into 21-bits, with the downside of only supporting a heap of no more than 4GB while pointers have a stride of 4KB. This approach might not be suitable when other algorithms are used to manage superblocks more flexibly, and introduce a limitation on the maximum size of the heap.

There are no indications that KMA-2 could be made compatible with AMD GPUs, given how this platform implements a very relaxed memory model. Other platforms appear to have adopted less relaxed memory models and therefore show no consistency issues when memory fences are carefully placed, but this approach does not work for AMD GPUs. However, this memory model is permitted by the current OpenCL specification[27].

Overall, we believe that the hypothetical KMA-2 has fair chances to be a better implementation than KMA-1 in terms of performance if more complex algorithms are used to maintain the state of the heap. Portability could be slightly improved on currently, but until the OpenCL specification

is clarified and the memory model is restricted, it is not possible to guarantee portability.

7.3 Notes on OpenCL portability

Although portability is one of the key features of OpenCL, we encountered several problems when developing our KMA. Improving portability between platforms by clarifying the OpenCL standard and unifying the implementations is not just beneficial to KMA, but to the entire OpenCL community.

First of all, as our experiments have shown, different OpenCL implementations can interpret the relaxed consistency model of OpenCL differently. This means that the optimisations done by compiler, although valid according to the OpenCL specification, leave room for speculation. For KMA this leads to malfunctioning on some platforms because optimisations break the semantics of the lock-free queue algorithm. A clearer specification of the memory consistency model of OpenCL, and the compilers to enforce it, can solve these functional portability issues. A further study towards the compatibility and alignment of functionality of different compiler front-ends is also needed to enable portability.

Secondly, correct usage of integer types like "uintptr_t" and "size_t" make it easy to write OpenCL code that is compatible between 32- and 64-bit platforms, but because the bitness of the targeted platform is not always equal to that of the host system, we still had to be cautious when allocating device-side memory objects containing pointers or variables of these types. OpenCL does not provide mechanisms to aid developers in handling these differences in bitness, other than a host-side function to query the bitness of the targeted device.

Finally, Intel's OpenCL compiler is quite picky about casting pointers back and forth to these integer types, which further complicates the process of writing portable code. The following code illustrates the problem we encountered with Intel's OpenCl compiler.

```
void function(char __global *base) {
        uintptr_t pointer_val;
        char __global *ptr;

        /* Example 1: does not compile with Intel's OpenCL SDK */
        pointer_val = (uintptr_t) base;
        pointer_val += 8;
        ptr = (char __global *) pointer_val;

        /* Example 2: compiles with Intel's OpenCL SDK */
        ptr = &base[8];
}
```

Both examples are functionally equal, yet the first example causes a casting error when compiled with Intel's OpenCL compiler, whereas the second example compiles without errors. In some cases the first form would be preferred, for instance when atomic operations are required that involve the address of an object. Atomic operations are generally defined for integer types and types like uintptr_t, but not for global pointers. When an atomic CAS operation returns a uintptr_t, this return value must be cast to a real pointer like shown in example 1 before it can be used. On the Intel platform this is impossible.

One idea that might improve interoperability between the different platforms is for the Khronos group to provide an OpenCL validation suite containing a series of test-cases that test the basic requirements for any OpenCL implementation. Besides being a good tool for developers to spot bugs and regressions, simple test cases could clarify the spirit of the standard and raise discussion when different parties disagree. The community driven Piglit project[1] successfully implements such a test suite for OpenGL compliance with support for OpenCL in place.

[1] http://people.freedesktop.org/~nh/piglit/

In addition, we believe a revision of the OpenCL synchronization primitives and their semantics, as well as potential new additions (in the form of global synchronization) will make the implementation of KMA-2 a lot easier, more robust, and portable.

Chapter 8

Conclusion

With the emerging of (massive-)parallel computation, several research projects towards efficient and usable parallel dynamic memory allocation have been conducted. For OpenCL no attempt has yet been made to implement a full featured heap allocator. In this light we have proposed KMA: a proof-of-concept dynamic memory allocator for OpenCL kernels.

The goal of this research project was to find an answer to the question: *how should a kernel-side heap allocator be implemented in OpenCL?* We aim to get an understanding of the place a heap allocator would take in the OpenCL platform, and of the complexity of such a tool. Our approach is to investigate two important aspects. Firstly we tried to understand the importance of a heap allocator to programmers by conducting a use-case survey towards parallel implementations of a variety of algorithms. Secondly we sought for the technical challenges of a heap allocator in OpenCL by implementing a prototype inspired by the state of the art.

Our thorough survey of applications (chapter 3) revealed that a specific set of algorithms require the creation of global non-regular data structures such as a linked-lists or trees. Dynamic memory management is a prerequisite for a lot of these cases to allocate the nodes in these structures. Along with the case of a C++-like object oriented kernel language, these cases led us to investigate and design KMA-1.

To answer our research question, we believe that a kernel-side heap allocator should be implemented in OpenCL by following a two-layered design, as proposed in chapter 4. This design is aimed at providing a familiar and universal interface to kernel programmers without the need to compromise on performance. The low level is an abstraction on top of the current OpenCL memory model to provide *malloc()* and *free()* routines to kernel developers. These APIs are available to any high-level data structure, which may then optimise the use of these APIs if the use-case permits. As a proof-of concept we designed the ArrayList top layer: a global array data structure that lets the threads in a work-group gather their memory requirements using a prefix-sum reduction to reduce the number of calls to *malloc()*.

Our contribution is the design (chapter 4) and implementation (chapter 5) of KMA. Our heap allocator design aims for portability and performance by simplicity. Experiments (chapter 6) have shown that KMA is functional and performs well when compared to Cuda's malloc and XMalloc[23]. The ArrayList implementation shows that when a use-case permits any restrictions, a layer on top of the KMA back-end can obtain even higher performance. KMA has a limited portability due to differences in the available platforms: NVIDIA Cuda and AMD APP for CPU correctly execute our test cases, but Intels SDK lacks support for 64-bit atomic operations while using 64-bit pointers, and AMD APP for GPU implements a more relaxed memory model leading to semantic problems with the used lock-free algorithm.

As we discussed in chapter 7, a future version of KMA could improve in three areas: reducing memory usage, increasing portability and extending features. Reducing memory fragmentation could be achieved either by the use of more advanced lock-free algorithms, or by extending the OpenCL standard to permit mutex-style locks. Portability can be improved mostly with unifications of the various OpenCL platforms, which probably requires some clarification and extension

of the current standard. There are several features that might extend the use of KMA, but implementing them often requires alterations to the standard or even hardware. These features should be considered based on user feedback.

We believe that the current KMA is a useful tool for all programmers that wish to experiment or work with irregular data structures. It offers a feature-complete heap allocator that performs similarly to the memory allocator found in Cuda. KMA allows programmers to write readable code with the benefits of a flexible heap allocator, and offers good potential for optimisations if the use-case permits.

Bibliography

[1] 1003.1 standard for information technology - portable operating system interface (posix) base definitions, issue 6. IEEE Std 1003.1-2001. Base Definitions, Issue 6, pages i –448, 2001.

[2] Parboil benchmark suite, 2010. http://impact.crhc.illinois.edu/parboil.php.

[3] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. Commun. ACM, 52:56–67, October 2009.

[4] J.-L. Baer and B. Schwab. A comparison of tree-balancing algorithms. Commun. ACM, 20(5):322–330, May 1977.

[5] Duane A. Bailey. Java Structures, pages 331–341. McGraw-Hill, 2002.

[6] Josh Barnes and Piet Hut. A hierarchical o(n log n) force-calculation algorithm. Nature, dec. 1986.

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

[8] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation, PLDI '01, pages 114–124, New York, NY, USA, 2001. ACM.

[9] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '02, pages 1–12, New York, NY, USA, 2002. ACM.

[10] Guy E Blelloch. Prefix sums and their applications. pages 35–60, 1990.

[11] M. Burtscher and K. Pingali. An Efficient CUDA Implementation of the Tree-based Barnes Hut n-Body Algorithm, pages 75–92. Morgan Kaufmann, Jan. 2011.

[12] Gregory Chaitin. Register allocation and spilling via graph coloring. SIGPLAN Not., 39:66–74, April 2004.

[13] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, pages 44 –54, oct. 2009.

[14] Intel Corp. AGP V3.0 Interface Specification, sep. 2002.

[15] Advanced Micro Devices. OpenCL Static C++ kernel language extension, dec. 2011.

[16] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In Proceedings of the 3rd international symposium on Memory management, ISMM '02, pages 163–174, New York, NY, USA, 2002. ACM.

[17] E. W. Dijkstra. A note on two problems in connexion with graphs. Numerische Mathematik, 1:269–271, 1959. 10.1007/BF01386390.

[18] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In The 40-th International Conference on Parallel Processing (ICPP'11), Taipei, Taiwan, September 2011.

[19] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Introduction to Parallel Computing. Pearson Education Ltd., 2003.

[20] Michael Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using dcas. In Proceedings of the twenty-first annual symposium on Principles of distributed computing, PODC '02, pages 260–269, New York, NY, USA, 2002. ACM.

[21] Lindsay Groves. Verifying michael and scott's lock-free queue algorithm using trace reduction. In Proceedings of the fourteenth symposium on Computing: the Australasian theory - Volume 77, CATS '08, pages 133–142, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

[22] Maurice Herlihy. Wait-free synchronization. ACM Trans. Program. Lang. Syst., 13:124–149, January 1991.

[23] Xiaohuang Huang, C.I. Rodrigues, S. Jones, I. Buck, and Wen mei Hwu. Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. In Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on, pages 1134 –1139, 29 2010-july 1 2010.

[24] Advanced Micro Devices Inc. Amd graphics cores next (gcn) architecture. white paper, jun. 2012. http://www.amd.com/la/Documents/GCN_Architecture_whitepaper.pdf.

[25] Free Software Foundation Inc. The GNU C Library, 2013. http://www.gnu.org/software/libc/manual/html_node/index.html.

[26] Byunghyun Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. Parallel and Distributed Systems, IEEE Transactions on, 22(1):105 –118, jan. 2011.

[27] Khronos. The OpenCL specification, nov. 2012.

[28] Jon Kleinberg and Eva Tardos. Algorithm design. Pearson, 2006.

[29] D Lea. A memory allocator, October 2000. http://g.oswego.edu/dl/html/malloc.html.

[30] Warren S. McCulloch and Walter Pitts. Neurocomputing: foundations of research. chapter A logical calculus of the ideas immanent in nervous activity, pages 15–27. MIT Press, Cambridge, MA, USA, 1988.

[31] Maged M. Michael. Scalable lock-free dynamic memory allocation. In Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, PLDI '04, pages 35–46, New York, NY, USA, 2004. ACM.

[32] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. Technical report, Rochester, NY, USA, 1995.

[33] N.N. R600-Family Instruction Set Architecture. Advanced Micro Devices Inc., jan. 2009.

[34] nVidia. nVidia OpenCL best practice guide, jul. 2009. www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf.

[35] NVIDIA. Nvidia geforce gtx 680, whitepaper. Technical report, 2012. http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf.

[36] nVidia corp. nVidia Cuda C Programming guide, apr. 2012.

[37] Ted Painter and Andreas Spanias. Perceptual coding of digital audio. In Proceedings of the IEEE, pages 451–513, 2000.

[38] A Penders and A. L Varbanescu. Accelerating graph analysis with heterogeneous systems. Master's thesis, Delft University of Technology, 2012.

[39] Mark Segal and Kurt Akeley. The OpenGL graphics system: a specification (4.3 Core profile), aug. 2012.

[40] Albert Silberschatz, Greg Gagne, and Peter Baer Galvin. Operating System Concepts, pages 286–288. John Wiley and Sons, Inc, 2005.

[41] Håkan Sundell and Philippas Tsigas. Lock-free deques and doubly linked lists. J. Parallel Distrib. Comput., 68(7):1008–1020, July 2008.

[42] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection, RAID '09, pages 265–283, Berlin, Heidelberg, 2009. Springer-Verlag.

[43] Andrea Vattani. k-means requires exponentially many iterations even in the plane. In Proceedings of the 25th annual symposium on Computational geometry, SCG '09, pages 324–332, New York, NY, USA, 2009. ACM.

[44] C.M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi gf100 gpu architecture. Micro, IEEE, 31(2):50 –59, march-april 2011.

[45] Shucai Xiao and Wu chun Feng. Inter-block GPU communication via fast barrier synchronization. In IEEE International Symposium on Parallel and Distributed Processing, IPDPS, 2010.

[46] E Young and F Jargstorff. nVidia Cuda video decoder - API Reference, aug. 2010.

Glossary

algorithm class basic pattern as described in "A view of the parallel computing landscape"[3]. 21

CAS Compare-And-Swap. 8, 9, 15, 16, 39–41, 48, 65, 73

critical section Part of code that may only be executed by one thread at a time. Protected by means of locking. 7

DFA Deterministic Finite Automatron. 25

FFT Fast-Fourier Transform[]. 29

FSM Finite State Machine. 25

GART Graphics Address Remapping Table, table used to map system memory to the GPUs address space, so that the GPU can access this memory. 13

GPGPU General Purpose Graphical Processing Unit. 5, 21

GPU Graphical Processing Unit. 5, 24, 29

kernel Function used as entry point in device-side OpenCL code. 14

lock Shared variable that is used by a routine implementing a mutex. 7, 73

lock-free Type of algorithm that does not require the use of mutual exclusion to protect its shared resources. Often makes use of atomic operations like CAS. 7

locking The act of reserving access to a shared resource or part of code (critical section), mutually excluding other threads. 7, 73

MIMD Multiple Instruction, Multiple Data. 7, 15, 18

mutex Mutual exclusion, protection mechanism that prevents concurrent access of shared variables. 7

NFA Non-Deterministic Finite Automatron. 25, 30

race condition Unpredictable behaviour when two concurrent threads try to alter the same shared variable at the same time. 7

semaphore see lock. 7

SIMD Single Instruction, Multiple Data. 7, 12

SIMT Single Instruction, Multiple Thread. 7, 8, 12, 14, 19

thread Running instance of a program on arbitrary hardware, processing a part of the workload. 7