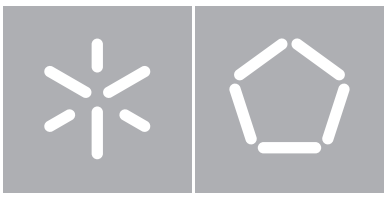
Universidade do Minho
Escola de Engenharia

Ricardo Daniel Queirós Alves

Distributed Shared Memory on Heterogeneous CPUs+GPUs Platforms

Outubro de 2012



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Ricardo Daniel Queirós Alves

Distributed Shared Memory on Heterogeneous CPUs+GPUs Platforms

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de
Professor Luís Paulo Santos
Professor Donald Fussell

Outubro de 2012

Universidade do Minho

Declaração RepositóriUM: Dissertação de Mestrado

Nome: Ricardo Daniel Queirós Alves

Nº Cartão Cidadão /BI: 13258703 Tel./Telem.: 933270434

Correio eletrónico: ricardoa@di.uminho.pt

Curso Mestrado em Engenharia Informática Ano de conclusão da dissertação: 2012

Área de Especialização: Computação Paralela e Distribuída

Escola de Engenharia, Departamento/Centro: Departamento de Informática

TÍTULO DISSERTAÇÃO/TRABALHO DE PROJECTO:

Título em PT : Distributed Shared Memory em Plataformas Heterogéneas com CPUs e GPUs

Título em EN : Distributed Shared Memory on Heterogeneous CPUs+GPUs Platforms

Orientadores Luis Paulo Peixoto dos Santos

Donald Fussell

Declaro sob compromisso de honra que a dissertação/trabalho de projeto agora entregue corresponde à que foi aprovada pelo júri constituído pela Universidade do Minho.

Declaro que concedo à Universidade do Minho e aos seus agentes uma licença não-exclusiva para arquivar e tornar acessível, nomeadamente através do seu repositório institucional, nas condições abaixo indicadas, a minha dissertação/trabalho de projeto, em suporte digital.

Concordo que a minha dissertação/trabalho de projeto seja colocada no repositório da Universidade do Minho com o seguinte estatuto (assinale um):

1. ☒ Disponibilização imediata do trabalho para acesso universal;
2. ☐ Disponibilização do trabalho para acesso exclusivo na Universidade do Minho durante o período de
   ☐ 1 ano, ☐ 2 anos ou ☐ 3 anos, sendo que após o tempo assinalado autorizo o acesso universal.
3. ☐ Disponibilização do trabalho de acordo com o **Despacho RT-98/2010 c)** (embargo___ anos)

Braga/Guimarães, 31 / 10 / 2012

Assinatura: Ricardo Daniel Queirós Alves

Acknowledgements/Agradecimentos

Apesar do processo de elaboração de uma tese de mestrado seja algo solitário por natureza, esta dissertação reúne contribuições de várias pessoas. Durante o meu mestrado contei com o apoio e confiança de imensas pessoas, sem as quais este projeto não teria sido possível.

Ao meu orientador, professor Luís Paulo Santos, agradeço a sua paciência, disponibilidade e as diversas discussões e contribuições que foram muito valiosas para este trabalho. *I would also like to thank my co-advisor, professor Donald Fussell, for the opportunity and trust invested in me.* Quero também estender este agradecimento aos professores Alberto Proença e José Nuno Oliveira que, apesar de terem contribuído de uma forma mais indireta nesta dissertação, assumiram um papel de destaque na minha formação ao longo de todo o meu percurso académico na universidade do Minho.

Aos meu colegas do *LabCG* agradeço o apoio e a disponibilidade para debater e trocar ideias. Mas acima de tudo, agradeço a amizade e o ambiente que criaram no laboratório que tornou todo o processo de elaboração desta dissertação muito mais agradável. Em especial, agradeço o contributo e apoio do engenheiro João Barbosa, que foram essenciais para a conclusão deste projeto.

Por último, agradeço aos meus pais, familiares e amigos o apoio e incentivo que me deram, não só neste último ano, mas ao longo de todo o meu percurso académico e pessoal. Um humilde obrigado para amortizar uma dívida que nunca vou conseguir pagar.

Resumo

Desenvolver aplicações para plataformas heterogéneas pode dificultar significativamente o processo de codificação, visto que o uso de dispositivos de computação diferentes significa ter que lidar com arquiteturas diferentes, modelos de programação e organização de memória diversos, espaços de endereçamento de memória disjuntos, etc. Este documento propõe que o processo de desenvolvimento pode ser simplificado ao virtualizar um ambiente de memória partilhada tradicional em cima de um sistema de memoria heterogéneo distribuído e expondo um modelo de memória unificado ao programador. O sistema de memória liberta o programador da gestão manual dos dados e permite o uso de memória dinâmica acessível por todos os dispositivos.

O sistema de memória proposto foi implementado e validado na *framework GAMA* usando três algoritmos para testar o sistema: SAXPY, simulação N-Body "all-pairs" e Barnes-Hut. Estes algoritmos foram usados para avaliar o desempenho e a escalabilidade da *framework* quando equipada com o sistema de memória proposto.

Os resultados mostram que, de uma forma geral, o sistema de memória melhorou o desempenho de todos os algoritmos. O sistema de memória provou ser mais útil em algoritmos com uma alta razão de computação sobre acessos a memória e especialmente em algoritmos irregulares ao melhorar também a escalabilidade. O alocador de memória paralelo mostrou óptimos resultados quando usado apenas no CPU, mas teve problemas na velocidade de alocação quando foram adicionados GPUs ao sistema.

Abstract

Developing applications for heterogeneous platforms can significantly complicate the coding process, since different processing devices mean different architectures, programming and memory models, disjoint address spaces and so on. This document proposes that the development process can be eased by virtualizing a traditional shared memory environment on top of the heterogeneous distributed system and exposing a unified memory model to the developer. The memory system frees the developer from having to manually manage data movements and allows the use of dynamic memory, accessible by all the devices.

The proposed memory system was implemented and validated on the *GAMA* framework using three algorithms to benchmark the system: SAXPY, all-pairs N-Body simulation and Barnes-Hut N-Body simulation. These algorithms were used to evaluate the framework's performance and scalability when equipped with the proposed memory system.

The results show that, overall, the memory system improved performance on all algorithms. The memory system proved most useful on algorithms with high ratio of computation over memory accesses by improving execution times and especially useful on irregular algorithms by improving scalability as well. The parallel memory allocator showed great results when used only on CPU, but had speed issues when GPUs were added the system.

Contents

List of Figures

List of Tables

Nomenclature

Cg	C for Graphics
CMP	Chip Multi-Processors
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
DSM	Distributed Shared Memory
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
GAMA	GPU And Multicore Aware heterogeneous framework
GCN	AMD's Graphics Core Next GPU architecture
GLSL	OpenGL Shading Language
HT	HyperTransport
ILP	Instruction Level Parallelism
IXP	Intel family of network processors based on ARM micro-architectures
MIMD	Multiple Instruction Multiple Data
NUMA	Non-Uniform Memory Access
OpenCL	Open Computing Language
QPI	Quick Path Interconnect
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SMP	Symmetric multiprocessing
SMT	Simultaneous Multi-Thread
TBB	Thread Building Blocks
VLIW	Very Long Instruction Word

Chapter 1

Introduction

Discontent is the first necessity of progress.

– *Thomas A. Edison*

Graphics processors (GPUs) were developed to accelerate the creation and manipulation of computer graphics. With the increasing popularity of video games these processors became quite common and powerful, enclosing great computation capability[28]. Recent architecture improvements on the pipeline programability and the introduction of general purpose programming languages like CUDA[30] (Compute Unified Device Architecture) and OpenCL[37] (Open Computing Language), made the GPU a popular processor in a vast array of areas varying from scientific simulation on high end super computer clusters to simple video decoding on common desktop computers.

The CPU (Central Processing Unit) was designed with a more generalist purpose in mind and was oriented to the execution of sequential code. For many years the strategy for improving CPU performance was based on increasing chip clock rates, but due to physical limitations, that is no longer a viable solution[38]. With this limitation and Moore's law[27] still in effect, the alternative was to increase the number of processing cores in a single die. This is a dramatic change in CPU architecture since one can no longer expect to see performance improving in sequential code by simply upgrading the CPU, the algorithms have to be rethought and recoded in order to take advantage of the new parallel nature of

the CPU.

With the new parallel orientation of the CPU and performance scalability no longer being transparent to the programmer, the algorithms must explicitly explore parallelism in order to take advantage of the new CPU parallel architectures. This presents an unique opportunity to broaden horizons and explore other parallel processors to implement the new algorithms. The GPU is an excellent candidate because, not only it is almost ubiquitous in modern consumer computers, but it also encloses great computational potential, with a peak arithmetic performance that far exceeds its CPU counterpart. Many state of the art super computers are also equipped with gpus as seen on the top500[1] list (June 2012).

However, one should not focus on which processor to use (CPU or GPU) but on how to use both simultaneously and fully take advantage of the machine potential. This is not always an easy task since, despite the new found parallel nature of modern CPUs, GPUs and CPUs are still very different (architectures, instruction sets, memory hierarchies, programming languages, etc.) and developing for these systems can be very challenging. To improve productivity when programing for heterogeneous CPUs+GPUs systems, new development tools are a fundamental requirement.

1.1 Distributed shared memory systems

In parallel programming two main programming models exist – shared memory and message passing. The shared memory programming model is the more straightforward of the two because it is a natural extension of the single processor programing model: the data resides in a memory pool that can be directly accessed by any thread at any given time. This model is generally common on tightly coupled processor machines, where all processor chips and processor cores share a common interface to the memory. This shared interface to the memory is however a serialization point and thus a scalability bottleneck.

In distributed memory systems, each processor (or group of processors) has its private memory and is connected to others processors by a high speed (low

[1] Top500 http://www.top500.org/lists/2012/06

latency high bandwidth) network. These systems do not have the same scalability problems as a shared-memory machine, since there are dedicated interfaces to memory and with a well designed interconnection network topology between processors or group of processors, this type of system can scale virtually indefinitely. The tradeoff for scalability is the increase in programming complexity. These systems generally support a message passing programing model where each thread can access data in its private memory directly, but has to request non-local data explicitly before accessing it. This requires the programmer to have a deeper knowledge of his program data requirements and to design the algorithms to minimise remote data accesses.

Since discrete GPUs have dedicated memory, general purpose computing languages such as CUDA or OpenCL require the programmer to explicitly move data between the (CPU) host memory and the (GPU) device dedicated memory, similarly to distributed memory systems. Recent improvements on GPU architectures and programming languages allow the programmer to access host memory from the GPU directly, so explicitly memory movements are no longer compulsory. This eases the programming effort, but sacrifies scalability because, much like shared memory system, serializes memory accesses with other devices on the machine. So in an heterogeneous system with GPUs, we are faced with the same problem as with a multi processor system – programming simplicity vs scalability – with the added complexity of the asymmetric nature of host memory and device memory, that have different sizes, bandwidths, clock frequencies, etc.

A way to circumvent the additional programming complexity of distributed memory systems is to use a shared memory abstraction layer on top of the message passing system – distributed shared memory (DSM) system. With the DSM, the physical disjoint memory spaces can be addressed as a single, logically shared, memory space. A software distributed shared memory system can be interpreted as an extension to the underlying operating system virtual memory. Similarly to the operating system virtual memory mechanism, the software DSM is completely transparent to the programmer.

1.2 Dynamic memory allocators

Another important step in bringing a familiar multi-core homogeneous environment to an heterogeneous CPU+GPU computing system is to enable the use of dynamic memory on these heterogenous systems as in a shared memory environment. Dynamic memory allocators are used in a variety of applications. The inability to use dynamic memory, poses a barrier to porting existing shared memory programs (that rely on dynamic memory) to heterogeneous systems.

Dynamic memory is already possible on accelerators like modern GPUs, but the data allocated dynamically by GPU threads are only accessible by those GPU's threads. The data is not visible by other GPUs' and CPUs' threads. The same happens with dynamically allocated memory by CPU threads. An heterogeneous dynamic memory allocator on an heterogeneous system should behave as a traditional memory allocator on a traditional system, all dynamic memory allocated by the system's threads should be accessible by any other thread.

1.3 Motivation and goals

The new found general purpose side of the GPU, started a new era in software development. This opened a lot of possibilities, but also introduced software developers to new challenges – new architectures, programming models, programming languages, etc. New parallel CPU architectures already put a hard strain on programing complexity and the introduction of GPU many-core architectures only increased the problem. The use of both CPU and GPU simultaneously only adds to the already existing intrinsic difficulties of parallel programming of both families of processors with disjoint memory spaces and different memory models, with different consistency models and synchronization primitives.

This document proposes that the programming burden of heterogenous platforms can be eased by delegating data management to a runtime memory system. This system should unify the disjoint memory address spaces and expose to the programmer a single memory model. This frees the programmer from having to explicitly deal with data transfers and use architecture specific data man-

agement primitives. By charging the memory runtime system with these tasks, also enables the opportunity of the memory manager to work with the runtime scheduler and dynamically optimize data transfers. The heterogeneous memory subsystem should not only improve programing productivity but also improve code portability with the minimum negative impact in performance.

The memory runtime system should also allow the programmer to reserve and free memory dynamically. Not only traditional multi-core systems allow the use of dynamic memory but several algorithms and applications rely on the use of dynamic memory. The necessity of enabling the use of dynamic memory is further confirmed by the recent efforts of bringing dynamic memory to GPUs [17][30]. Constructing a familiar and flexible memory environment on heterogeneous systems must include a dynamic memory manager.

The proposed heterogeneous runtime memory system allows: (1) The programmer to be data location agnostic – no explicit data movement requirement by the programmer or even be aware of whether the accelerators have dedicated memory or not. (2) Improve code portability – the systems exposes a unified architecture independent memory model and memory operations. (3) Runtime data management optimizations – by optimizing data movement at runtime, the system can adapt to the system characteristics at real time. (4) The use of dynamic memory – any thread on a system can allocate and free memory visible by the whole system, independently of the device where the threads reside. The dynamic memory manager also has the same four characteristics that other modern memory allocators have: fast alloc/free operations, thread scalability, false share avoidance and low fragmentation.

The focus of this work is set on CPU(s) and GPU(s) heterogeneous systems, but the model should be flexible enough to suport other systems with aditional classes of acelarators, like FPGAs, DSPs, etc. The heterogenous memory runtime system was implemented and tested on the *GPU And Multicore Aware* (*GAMA*) heterogeneous framework.

1.4 Document organization

Chapter 1 gives a brief introduction to heterogenous programing, the problems software developers face and possible solutions. Chapter 2 presents an overview of the modern CPU multi-core architectures and GPU many-core architectures; and gives some background on existing programing models, distributed shared memory systems, parallel memory allocators and heterogeneous frameworks. Chapter 3 specifies in detail the problem and the proposed approach on how to tackle it. Chapter 4 details the proposed heterogeneous runtime memory system – the memory model, heterogeneous distributed shared memory and heterogeneous parallel memory allocator. Chapter 5 contais the experimental validation of the proposed memory system on the *GAMA* heterogenous framework. Chapter 6 gives a final overview on the project, summarizing and criticizing the project and commenting on future work.

Chapter 2

Background and related work

If I have seen further it is by standing on ye sholders of Giants.

– Isaac Newton

Heterogeneous systems can be very diversified and include a great variety of accelerators. The focus of this project however resides solely on CPUs and GPUs only heterogeneous systems. In this chapter some background is given on the multi-core CPU architectures and many-core GPU architectures. The history and characteristics of software distributed memory systems and parallel memory allocators is described as well. The chapter ends with an overview and critical analysis of existing heterogeneous frameworks' memory systems.

2.1 Parallel architectures

Different problems and objectives spawned several types of processor architectures. Given the broad nature of problems that processors need to target, two general approaches are taken – producing an all-purpose processor capable of addressing all problem domains by sacrificing some performance over flexibility; or producing domain specific processors, capable of tackling only a set of problems but being very efficient at it. Modern machines are normally equipped with a general purpose processor (CPU) and one or more domain specific processors

(accelerators). One of the most popular and ubiquitous accelerator is the one designed to tackle real time computer graphics (GPU).

2.1.1 CPU multi-core architecture

For many years the center of a computing device was a single-core architecture processor, responsible for executing instructions sequentially one at a time. Moore's law predicted that the number of transistors in an integrated circuit could be doubled inexpensively every two years [27] and that dictated the evolution of the CPU, resulting in higher clock speeds. However, constraints in the current processor manufacturing technology made this evolution path unsustainable. Due to power consumption and heat dissipation restraints [38], the CPU clock rates would eventually stagnate and alternatives were explored to complement (and eventually replace) the increase in clock rates as the main improvement on newer CPUs.

With power limitations and Moore's Law still in effect manufacturers used the extra transistors to increase the number of functional units by placing redundant arithmetic and control units in a single die. In order to explore these units new techniques were explored to extract parallelism at the instruction level (ILP - Instruction Level Parallelism). Depending on manufacturer and processor class different sets of ILP techniques are implemented – Pipelining, Very Long Instruction Word (VLIW), Single Instruction Multiple Data (SIMD), etc.

The ILP techniques have the advantage of being used with little to no change in the sequential source code written for traditional sequential processors, since the processor will explore parallelism automatically and new parallel instructions will be introduced in the program by the compiler simply by activating compilation options. But the slow increase on main memory clock speeds when compared to the CPU [40], made memory accesses increasingly costly and optimizations introduced by ILP techniques produce poor results even with larger and faster caches.

The solution to this problem could no longer pass through transparent parallelism introduced by ILP techniques but through explicit parallelism. By dividing an operation into several parallel tasks (computational threads) the processor

could hide a thread's memory access latencies by executing instructions of another thread until the data of the previous thread is available, thus keeping functional units occupied – this technique is called Simultaneous Multi-Thread (SMT). SMT efficiently helps hide memory accesses latencies but also introduces complexity for the programmer since parallelism must be expressed in the algorithm/code explicitly.

One way of increasing performance is to multiply the number of computational units. This can be done on a single computational node by increasing the number of CPUs per node – this technique is know as Symmetric Multiprocessor (SMP). SMP is an old technique used when the technology does not mach the computational requirements of the task. Chip Multiprocessors (CMP) derive directly from SMP but places the processors (cores) in a single die (processor package) – multi-core architecture. CMP improves on SMP since communications between processor cores are done inside the chip, which is faster, more energy efficient and through the use of shared caches, it's easier to keep the memory coherent. This shift to paralel architectures stimulated the creation of new tools like OpenMP[11], TBB[32] and Cilk[8] to help the software developers better explore the processor potencial.

With the new tendency of placing multiple cores/processors in a single computational node the traditional communication mechanism between processors and main memory had to be rethought. Since the communication channel to the memory is shared by all cores/processors, by increasing the number of cores/processors in a node, the number of bus conflicts also increases. With few cores or processors this problem is neglectable, but with the new tendency in increasing the number of cores per processor, this communication model is unsustainable. The alternative comes in the form of NUMA (Non-Uniform Memory Access) architectures. In a NUMA architecture, groups of processors/cores have a local memory and a dedicated communication channel to it. In order for a processor to access non local memory addresses it has to request the data from other processor's local memory.

The communication between processors/cores in the case of Intel processors is done using QPI[19] (Quick Path Interconnect) or HT[18] (Hyper Transport) in the case of AMD. Depending on the processor communication network topology,

a processor might not have a direct communication channel between all the other processors and a remote memory access might have to send its request through several processors before it reaches the desired one. This adds latency to data requests since the farther away (request wise) the data is, the slowest it is to reach. NUMA architecture solves the scalability problem but introduces a new one: memory access times are not uniform anymore, local memory access times are faster than a non local access. A diagram of a NUMA architecture system with four processors can be viewed in figure 2.1.

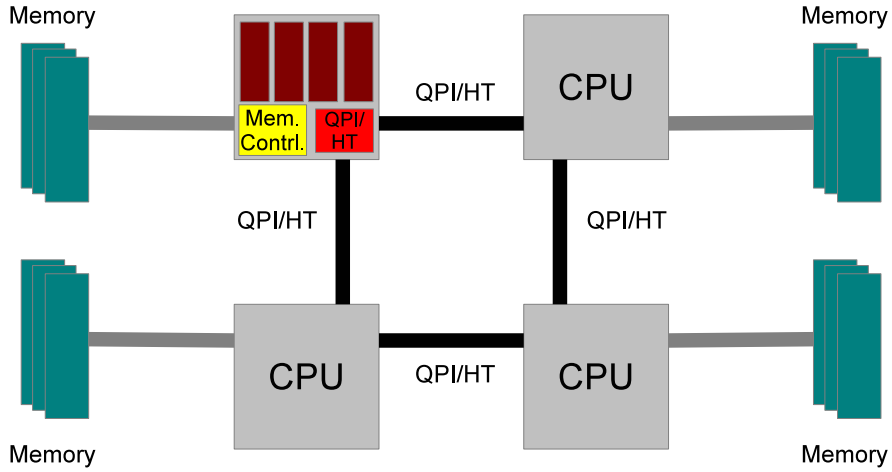


Figure 2.1: NUMA architecture in a four multi-core processor topology. Each processor package has a dedicated channel to its local memory and communicates with other CPUs through QPI/HT to access non local memory.

2.1.2 GPU many-core architecture

CPU architectures target a variety of different problems and so evolved with this objective in mind, spawning very versatile architectures. More specific domains produced different processors with very specific orientation architectures. One of these domains was computer graphics and the class of processors produced for this specific domain was the GPU.

GPUs are optimized for throughput instead of speed. A GPU architecture sacrifices complex control logic (instruction prefetching, branching prediction,

etc.) and large caches (to hide latency) in order to obtain a simpler computing core using less transistors, allowing more cores to be packed in a single die. By having lower clock rates, it reduces the energy consumption of the chip, making heat dissipation easier, which in turn allows more cores per chip.

This type of architecture appeared to solve the realtime rasterization problem. Since rasterization is an intrinsically parallel task, the GPU architecture had (and still has) the same parallel nature. From early on, these types of processors enclosure great computational capability, but they were used exclusively for computer graphics acceleration. It wasn't until the late 90's that the first GPUs with programable pipelines appeared. Initial attempts to use GPUs as general purpose used languages borrowed from computer graphics such as Cg[25] (C for Graphics) or GLSL[34] (OpenGL Shading Language). But with still low programmable capacity and no floating-point arithmetic capability, the use of the GPU for general purpose computing was limited.

Floating point arithmetic capability emerged in these processors in the form of floating-point color buffers in 2002[28]. The purpose of this was to produce richer and more complex computer graphics but it opened the door for general purpose computing on the GPU. Market competitiveness and demand for better graphics stimulated further improvements and modern GPU architectures introduced more programability on the previously fixed pipeline and improved floating point support each time closer to IEEE754 specification.

The introduction of Nvidia's Compute Unified Device Architecture (CUDA)[23] marked an important step in GPGPU (General Purpose GPU), introducing a general purpose language for GPU programming (C for CUDA), and replacing industry specific terms like shaders, texels, pixels, etc. with more conventional terms like cores, threads, cache, etc. Other GPGPU programing languages emerged like the Khronos Group OpenCL[37] or Microsoft's DirectCompute[9].

Nvidia Fermi's architecture[41] (and it's recent iteration, Kepler's architecture) is composed of simple computing cores called Stream Processors (SPs) or CUDA cores. Each CUDA core is a pipelined scalar processor capable of single precision integer or floating point operations. Double precision arithmetic is achieved pairing two single precision cores. The SPs are grouped into Stream Multi-Processors (SMs). High end versions of Nvidia Fermi processors have 32

SPs per SM. Like the SPs, the number of SMs per chip varies, generally high end versions have more SMs than lower versions of Fermi processors (GF110 GeForce GTX 580 has 16 SMs). Each SM has load/store units (16 on GF110) and special functional units (4 on GF110) that allow transcendental operations. An individual SM also has register memory and fast memory. This fast memory is used as L1 cache and as an explicitly addressable memory (referred to as shared memory). This memory is shared by all CUDA cores within an SM. The chip also has L2 cache, that is shared by all SMs on the chip and serves as the interface to the chip outside memory be it device or host memory. The details are illustrated in Figure 2.2

Figure 2.2: Nvidia Fermi GF100/GF110 architecture. Chip diagram on the left and SM diagram on the right. (Source Nvidia)

The execution model of each SM can be viewed as a 32-wide SIMD processor. The threads are grouped into blocks (thread block) and the blocks are assigned to SMs for execution. The execution model is not truly SIMD, since SMs deal with conditions differently than an SIMD processor would. A SM does allow divergence between threads (but with a considerable penalty). The model is called SIMT (Single Instruction Multiple Thread). Threads can only be synchronized with other threads from the same thread block, even if two blocks are assigned to the

same SM. However, CUDA has cache L2 atomic operations that can be used for synchronization between threads of different blocks.

2.2 Distributed shared memory

There are several design considerations that have to be taken into account when developing and implementing a distributed shared memory system[29] [12] [33]. These considerations can be divided into the following groups:

- **Structure and Granularity** – Structure refers to the layout of the data in memory (linear array of words, data objects, language types, etc.) and granularity refers to the size of the unit of sharing (byte, word, page, etc.).
- **Coherence protocols and Consistency models** – When there is data replication, a coherence protocol guarantees that all nodes view all memory positions in the same way. A consistency model determines the conditions under which memory updates will be propagated through the system.
- **Scalability** – Distributed Shared Memory systems can have their scalability limited by central bottlenecks (e.g sharing the same communication bus) and global operations (e.g. synchronization operations).
- **Heterogeneity** – DSMs over heterogeneous systems are intrinsically more complex. An heterogeneous DSM system might include subtle differences (e.g. different memory size or cache levels) to more technical challenging differences, like different processor architectures, instruction set architectures (ISA), memory hierarchies, etc.
- **Data location** – When a thread requests data that is not in its local memory, the DSM must support mechanisms to find and retrieve the requested data.
- **Replacement strategy** – When a cache is full, replacement strategies are required to decide which cache lines should be replaced to free up space for the new incoming data.

- **Trashing** – Trashing happens when data that cannot be replicated is needed by more than one device at a time or the data is being often written by one device and read by other.

2.2.1 Structure and Granularity

Both these aspects of DSMs are closely related. Most DSMs do not structure their memory[29] and use a simple memory pool. This way the DSM is faster (since it introduces less computational overhead) and more flexible (since it has a lower level view of the memory). Programs that use shared memory typically explore data locality, i.e. a thread is likely to address spatially close address spaces in the near future, so the use of bigger sharing units might reduce communication overhead. On the other hand, a large "page" increases the probability that more than one thread might need the same page at the same time and so increases the probability of false sharing. Having the memory structured can help reduce these problems but also introduces complexity. There isn't a common page size on DSMs with unstructured memories, implementations have the page sizes varying from 16Bytes (Dash[20]) to 8KBytes (Mermaid[43]). Examples of DSMs with structured memories are Munin[6] which structures memory into objects (integers, arrays, structures, etc.) and Linda[1] that uses a database like type of structure.

2.2.2 Coherence protocols and Consistency models

"A memory consistency model is the logical sum of the ordering of events in each processor and the coherence protocol" [35]. There are two kinds of coherence protocols: write-invalidate and write-update[36]. 1) Write-invalidate – after a piece of data is written locally, all copies of the same data on other nodes are marked as invalid; 2) Write-update – after a piece of data is written locally, all other copies of the same data are updated before any access to it is allowed. The write-invalidate protocol allows data writes to be made locally (local update) but data accesses by other nodes to the same data will always result in a cache miss. The write-update protocol on the other hand will not produce cache misses but each data write will always be a global update. Due to the cost of communication, most DSMs implement a write-invalidate coherence protocol. The most natural

and intuitive model is the Strict consistency model in which the programmer can expect the most recently written value after a read operation. However, such a strict model can have a severe impact in performance. Because of performance considerations distributed shared memory systems implement more relaxed memory consistency models. Figure 2.3 represents several consistency models in an hierarchical way.

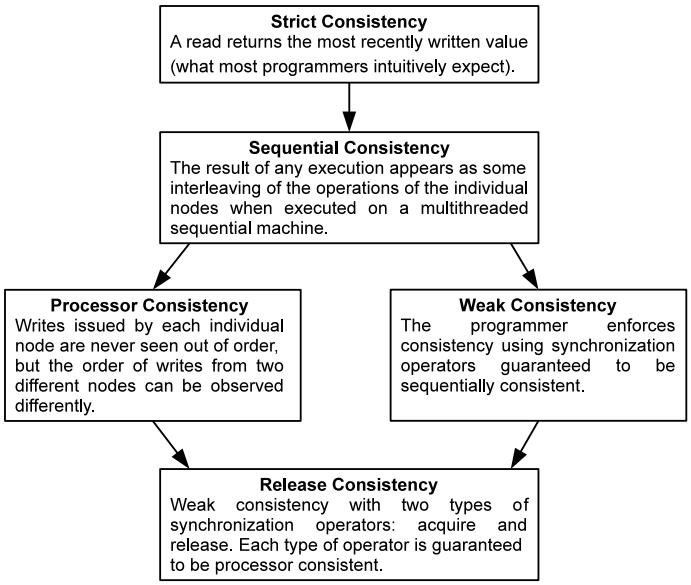


Figure 2.3: Intuitive definitions of memory coherence. The arrows point from stricter to weaker consistencies (Remastered version of original in [29])

2.2.3 Scalability

Most work about software DSMs and implementations rely on outdated architectures (single-core processors and a single shared bus across processor packages). Modern architectures have dedicated communication channels for different processors with integrated memory controllers (NUMA architecture) in a structure where each processor has its own private bus for its "local memory". This reduces the scalability bottleneck but introduces complexity since memory accesses latency times are no longer constant. Moreover, despite each GPU device having its local memory, it's not possible to directly access memory of a device from

a different one, i.e, all communication between devices must pass through main memory. Depending on the architecture, a GPU device might have to share a memory communication channel with some or all other GPU devices.

2.2.4 Heterogeneity

Distributed memory systems over heterogeneous systems are intrinsically more complex than homogeneous ones. A heterogeneous system might include subtle differences (e.g. different memory size or cache levels) to more technical challenging differences, like different processor architectures, instruction set architectures (ISA), memory hierarchies, etc. Some of these problems do not apply to CPU-GPU computing. Using a common combination of CUDA GPUs and IA-32 CPUs as an example, the CPU-GPU heterogenous system share the same programing language (with small differences), same data representation (e. g. endianness), etc. But other heterogeneous problems are still relevant: (1) Disjoint address memory spaces; (2) Different ISAs and therefore different execution binaries; (3) Computational capability, for example, limited support for conditional branching and no support for recursion on the GPU; (4) Different Memory hierarchy, with the introduction of CPU alien concepts such as explicit cache (shared memory) or local memory; (5) Different memory interfaces with unique capabilities to lower access latencies such as memory coalescing. There aren't many implementations of DSMs that support heterogeneous environments and most do not support modern heterogeneous architectures like multicore and many-core heterogeneous machines.

2.2.5 Data location

When a thread requests data that is not in its local memory, the DSM must support mechanisms to find and retrieve the requested data. Systems where there is no data migration or replication this can be trivially done, since data resides only and always in the same centralized location. However, if data is allowed to migrate and there is replication, the solution its not so easily achieved and several approaches to the problem can be considered. These solutions can be divided into to two groups[21]: Centralized approaches and Distributed manager

approaches. The Centralized approaches are simpler, but can overload the node with data accesses and slow down the entire system. Distributed approaches scale better but are significantly more complex.

2.2.6 Replacement strategy

Replacement strategies used in most DSMs are similar to the ones used in caching mechanisms. There are several replacement strategies but the more common are the least recently used (LRU) (or some approximation) or random replacement (RR). Distributed shared memory systems can also have additional information for each page that classifies them by type (e.g. shared, read-only, private, etc.) and in turn are used in the replacement policy. A distributed caching system would give evict priority to a shared read-only page over a privately owned page, since the later would have to be written back in main memory in opposition to a shared read-only page that could simply be erased. These page types cannot be the only factor when deciding which page to replace, because a read-only page, despite being easy to replace, could be accessed several times by that thread and not having it in cache means that it needs to be fetched from main memory.

2.2.7 Trashing

Trashing is a common problem among DSMs and strategies to reduce trashing generally revolve around some form of data replication. Munin allows the programmer to associate types to the shared data (private, migratory, read-only, etc.). But not only is this only possible on a DSM with a structured memory but also requires the programmer to specify the type of data (it's not a transparent process). Mirage[15] uses a dynamically tuned parameter on the coherence protocol that determines the minimum amount of time a page stays at a node (Δ). When a node writes on a shared page, that page remains writable on that node for a minimum of Δ time. Mirage dynamically analyses access patterns to tune the Δ value.

2.3 Dynamic memory on parallel architectures

One of the recurring concerns when dealing with parallel architectures is scalability and traditional memory allocators, designed to work on sequential machines, tend to under perform and scale bad on parallel architectures. Modern memory allocators were designed to address this problem. Several work exists on parallel memory allocators like Hoard[7], Ptmalloc[42] and Michael's allocator[26]. All these memory allocators use different lock or almost lock free algorithms to deal with the scalability problem.

Hoard[7] uses multiple processor heaps in addition to a global heap. Each heap contains zero or more superblocks. Each superblock contains one or more blocks of the same size. Statistics are maintained individually for each superblock as well as collectively for the superblocks of each heap. When a processor heap is found to have too much available space, one of its superblocks is moved to the global heap. When a thread finds that its processor heap does not have available blocks of the desired size, it checks if any superblocks of the desired size are available in the global heap. Threads use their thread ids to decide which processor heap to use for malloc. For free, a thread must return the block to its original superblock and update the fullness statistics for the superblock as well as the heap that owns it. Typically, malloc and free require one and two lock acquisitions, respectively.

Michael's allocator[26] recicles some of Hoard's high level structures (heaps, blocks and superblocks), reorganizes them slightly and replaces any locking mechanism with atomic operations (atomic CAS). This makes the allocator completely lock free. Each heap contains superblocks and each superblock is divided into same size blocks. Superblocks are classified and grouped into different size classes depending on their block size and each size class contains several processor heaps. XMalloc[17] ports the models introduced by both CPU exclusive Hoard and Michael's allocators to the GPU. XMalloc is a GPU parallel memory allocator implemented in CUDA. It follows a similar data structure to Hoard's superblocks and borrows the atomic CAS based lock-free idea from Michael's allocator.

2.4 Heterogeneous frameworks memory systems

With the popularity of the GPU for general purpose computing increasing recently also did the interest in developing tools to aid developers to use not only the GPU but both GPU and CPU. In an effort to ease the software development process in heterogeneous architectures, some frameworks arose (including the *GAMA* framework). These heterogeneous frameworks have different approaches and propose different programming and memory models. Of the several existing heterogeneous frameworks three stand out by their different approaches to data management: Harmony[13], Merge[22] and StarPU[4].

Harmony is a runtime system that offers a programing and execution model for heterogeneous systems. The objective is to simplify the programing of auxiliary accelerators like GPUs, FPGAs (Field-Programmable Gate Array) and Intel IXPs[1] without sacrificing performance. The programmer writes a collection of compute kernels that are dynamically assigned to available computing resources. The execution model is similar to out-of-order scheduling and execution of instructions on a superscalar processor – kernels are scheduled for execution depending on availability of data (data dependencies) and resources (accelerators). Each kernel only computes data on its own address space, which means that the same kernel cannot run simultaneously on different processors.

Merge is a framework for multi-core heterogeneous systems. The framework offers a programing language (Merge high level language) that uses the map-reduce patterns. The map-reduce enables automatic parallelization of the code and allows the framework to create independent work units to be assigned to processor cores. Merge uses EXOCHI [39] to implement the API for the various heterogeneous processors. EXOCHI can also be used by the programmer to extend Merge to others heterogeneous architectures. This framework is one of the most similar to a SMP (Symmetric multiprocessing) system in terms of memory operations, but the drawback is that it needs all the devices to share the same memory pool. This narrows the type of devices it supports and excludes the dedicated GPUs, the most popular accelerators today.

StarPU is a runtime system with dynamic task scheduling. A task is a set

[1] Intel family of network processors based on ARM micro-architectures.

of architecture specific implementations of the same computational kernel. Each task is then assigned to the available heterogeneous processors by the scheduler. StarPU implements a virtual shared memory with a relaxed consistency model with caching capability[3]. This framework is the more complete in terms of data management. It allows the access of data by any device on the system as in a shared memory environment and it moves data and keeps memory consistency automatically[2]. The framework uses the GMAC[16] system to manage data.

The GMAC system has similar characteristics as the proposed DSM – reduced programming effort and improved code portability by freeing the developer from having to deal with data movements explicitly. However, it has some shortcomings: Despite the CPU being able to access any program variable, (1) code running in the GPU cannot access variables hosted in CPU memory and (2) code running in the GPU cannot access variables hosted in other GPUs. Also, it does not support dynamic allocation and deallocation of memory (visible by all computing devices) by code running on GPUs.

2.5 GAMA framework

The *GPU And Multicore Aware* (*GAMA*) framework is a framework designed to extract performance and simplify the coding process of heterogenous computing environments with GPUs and other classes of accelerators. The framework unifies the independent programming, execution, and memory models of the different computing devices into a single one. This not only simplifies the coding process but also makes the source code independent of the underlying hardware, improving both coding productivity and portability. The framework is still in development and currently only supports systems with x86-64 processors and CUDA capable GPUs.

To implementing an algorithm, the programmer has to code at least two methods – the kernel and the dice method. This is identified as a job. The *GAMA* kernel works in a similar way as a CUDA or OpenCL kernel, it defines which data each thread will manipulate and how that data will be handled. The dice method defines how the data input domain will be divided across the parallel tasks and how many tasks will be generated from the kernel. It is similar to the

way we can define the number of threads and the granularity of each thread in OpenMP or the thread organization topology (grid and block sizes) in CUDA, but more flexible.

The framework builds individual binaries from the kernel targeted at the different computing device's architectures in the system and generates tasks from the dicing method. The runtime scheduler then dynamically assigns the tasks to different computing devices and executes the appropriate binary automatically. The programmer can implement an algorithm as a collection of jobs and tasks from different jobs can run in parallel or can be explicitly sequentialized by a synchronization primitive. The primitive guarantees that all tasks from the previous job have finished by the time tasks from the next job start.

Chapter 3

Problem Statement and Thesis

Simple things should be simple; complex things should be possible.

– Alan Kay

Heterogeneous systems introduced new programming challenges and aggravated existing ones. In this chapter, some of the problems with an asymmetric distributed memory system typical of heterogeneous environments will be presented and analyzed. In the end a solution for these problems is proposed and justified.

3.1 Problem

Data management is a significant and difficult problem when programming heterogeneous systems since correct data placement is crucial for achieving fast execution times. When dealing with these kind of systems the programmer is faced with several problems regarding data and memory:

- Explicit data movement – Accelerators have dedicated memory and, since generally the device cannot access host memory and vice versa, the programmer has to explicitly move the data to and from the devices. This requires the programmer to have a deep knowledge of his program data requirements and to explicitly code his algorithm in an effort to accommo-

date those data requirements with a minimum impact in performance due to memory copy latencies.

- Poor code portability – With a specific accelerator comes a specific memory model and set of primitives to manage data movement and synchronization (intra and inter devices). If the program has to be ported to another architecture, the algorithm has to be recoded to incorporate the new architecture specific memory model and primitives.

- No opportunity for runtime optimizations – By programming a device directly, the data movement operations are fixed and hardcoded to the program upon compilation. This leaves no room to dynamically "measure" the state of the system and decide the best option regarding data management. With no runtime help, the user is confined to a "trial and error" approach to determine the best way to move data around with the least performance impact. This also poses a potencial problem when adding new devices since it requires the programmer to repeat the "trial and error" process to find the best option on the new system with additional computing devices.

- Use of dynamic memory – Regardless of the programming model, accelerator code follows a simple pattern: copy data to the device; execute the device code on the that data; copy data back from device. The use of dynamic memory is generally not included in the model. Recent improvements on GPU architectures and programming models allow the programmer to allocate and free memory inside of the device, but this memory is not accessible by all cores/threads on the device and certainly not by other devices or CPU.

All these problems increase coding complexity and require the programmer to be familiar with the architecture and its primitives. This can have a significant impact in coding time and decrease productivity. This can even be more problematic if the code has to be ported to other architectures, which generally requires the algorithm to be recoded to incorporate the new architecture specific memory models and primitives. Existing heterogeneous frameworks try to tackle

some of these problems, but assume an easy and static data division (no data dependencies between devices and no dynamic memory required) or avoid the issue altogether by requiring the use of accelerators that don't have dedicated memory and share the host memory. StarPU stands out as an exception in this regard by acknowledging the importance of data management in code performance and offers automatic data movement and dynamic management of data dependencies. Unfortunately GMAC, StarPU's DSM, does not offer support for remote data access by the GPU threads, nor the use of dynamic memory.

3.2 Thesis

We propose a novel software memory system model for heterogeneous parallel machines that (1) unifies the disjoint address spaces of host main memory (CPU address space) and accelerator dedicated memory (GPU address space); and (2) allows the use and allocation of dynamic memory by all devices, i.e. each device can reserve and free memory visible by all devices on the system.

We argue that to simplify the programming effort and improve code portability on heterogeneous computing systems, the use of a distributed shared memory system with suport for dynamic memory is an important step. The heterogeneous memory system also follows a unified, architecture independent memory model. This essentially virtualizes a more familiar shared memory system on top of an asymmetric distributed memory system. The heterogeneous memory system allows:

- The programmer to be data location agnostic – No explicit data movement necessary. The runtime memory system will detect memory dependencies and move the data accordingly. The programmer manages and accesses data in the same way as in a shared memory system.

- Better code portability – The system unifies the different devices memory models into a single architecture independent memory model. The system also exposes a single set of data management and synchronization operations. The programmer will not have to know or use any of the underlying

architecture specific data management primitives. This allows the code to be ported without any modifications.

- Runtime data management optimizations – By delegating the data movement operations to the runtime system, it is possible for the memory system to dynamically optimize data movement for the current system characteristics and coordinate with the runtime scheduler to improve data reutilization and minimize data movement.

- Use of dynamic memory – The memory system allows any thread by any device to allocate memory, visible and accessible by any other thread on the system, at any given time. This helps bring the capabilities of heterogeneous systems closer to the ones of a traditional shared memory (CPU only) system.

The heterogeneous dynamic memory allocator follows the same four characteristics as a modern parallel memory allocator:

- Speed – An allocation/free operation should perform as fast as a sequential memory allocator. In the case of the heterogeneous memory allocator, these operations should be as fast as the (CPU only) parallel allocator.

- Scalability – The memory operations (alloc/free) should scale with the number of processors on the system i.e. the alloc and free operations execution time should remain constant, independently of the number of processors.

- False share avoidance – The allocator should avoid the sharing of memory pages in which different processors have data in the same memory page.

- Low Fragmentation – Heavy fragmentation can lead to under use of total available free memory and a significant decrease in performance of allocation and free operations.

Chapter 4

Design and Implementation

Software is not limited by physics,
like buildings are. It is limited by imagination, by design, by organization.
In short, it is limited by properties of people, not by properties of the world.

– Ralph Johnson

The proposed heterogeneous memory system can be divided into two main components – a DSM and a dynamic memory manager. The DSM maintains a virtual memory pool, shared across all computing devices, and the dynamic memory manager allows the dynamic allocation and deallocation of memory in this pool. In this chapter, the proposed memory and programming model will be presented together with the design of the heterogeneous DSM and memory manager.

4.1 Programming and memory model

Traditionally when using accelerators, the programmer declares data objects that are associated with a co-processor. The data is then moved to the specific co-processors and one or more kernels are executed on that data. An example of this model can be viewed in Figure 4.1. The memory space must be allocated in both the host and the device. Then the data is copied to the device, the kernel

is executed and after the execution, the data is copied back to the host. If using multiple devices, this process must be repeated for each device.

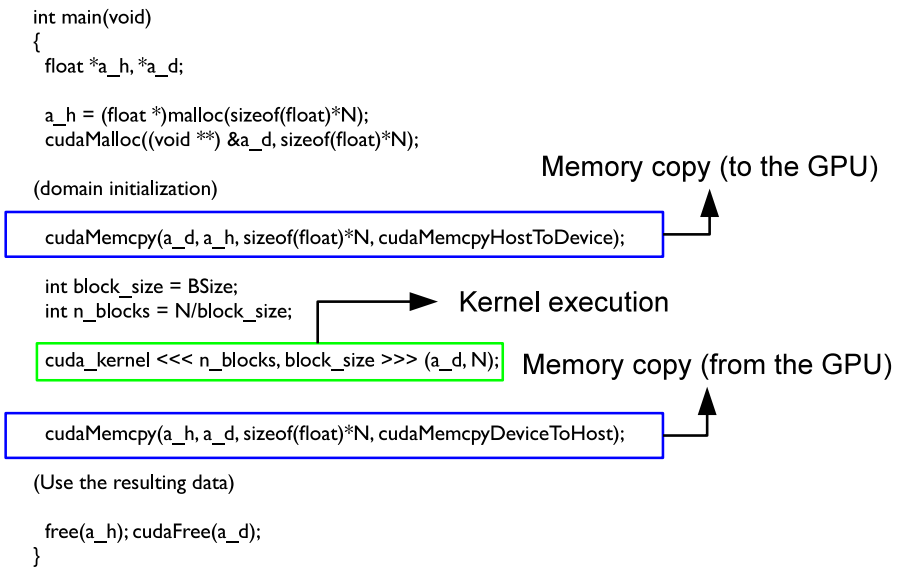


Figure 4.1: Simple example on how to execute a kernel in a single CUDA GPU

With a DSM there is no need to explicitly request memory on the different devices. Programmers have only to allocate data and run a single kernel (a job in *GAMA* terminology). The runtime system assigns the processor specific kernels (generated from the user's architecture independent kernel) to the respective devices and manages data transfers automatically. Figure 4.2 shows an example of running a job in the *GAMA* framework with the DSM system. The programmer only has to allocate the necessary data and execute the job. Since the execution is done asynchronously, the programmer must also use a synchronization barrier if needed.

The system exposes a simple API for allocating/freeing memory – an alloc and dealloc methods (Table 4.1). The programmer only has to specify memory size and the memory properties required – *HOST*, *DEVICE* or *SHARED*. Memory allocated with the *HOST* property will only be accessible by the host. Memory allocated with the *DEVICE* property will only be accessible by the computing device that allocated it. Finally, memory allocated with the *SHARED* property

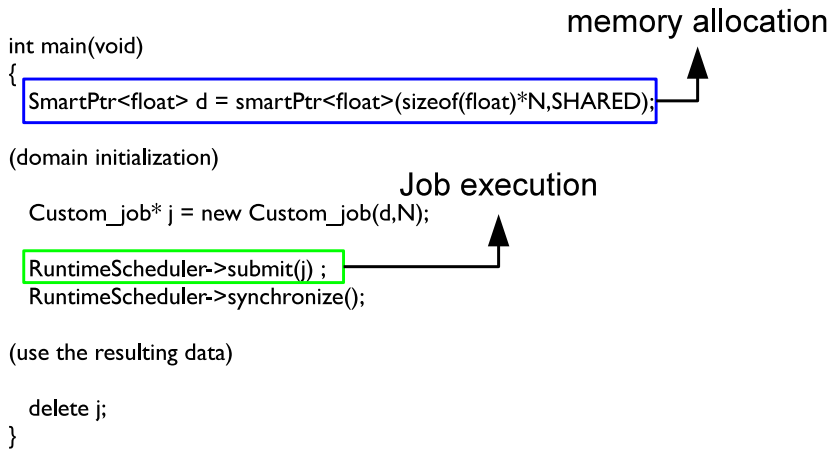


Figure 4.2: Simple example on how to execute a job in *GAMA* regardless of the number of accelerators

will be accessible by all computing devices on the system. This distinction can be useful since some data might only be necessary in the current thread/device scope, and specifying this fact explicitly allows the memory runtime system to avoid unnecessary data tracking, improving performance.

When coding the algorithm, aside from the kernel and the dice method required by the *GAMA* framework, the programmer has to specify two additional data methods. These methods allow the memory runtime system to identify the different tasks' data requirements and manage data coherence accordingly. One of the methods identifies the read-only data (ReadOnlyElems(void)) and the other the read/write data (ReadWriteElems(void)) for each thread. These methods must return a list of the elements that will not be modified and the ones that will (respectively). Since the only requirement of the methods is for the output to be a list of elements, the body of the methods (i.e how the list is created) can be as complex as the programmer needs them to be.

The DSM follows a weak consistency model and tasks generated from a job are scheduled arbitrarily. Order between jobs can be enforced by using the synchronization primitive "synchronize()". The programmer can also manage access to shared resources within a job using atomic operations such as "atomicCAS", "atomicAdd" and "atomicSub". The full list and description of all primitives can

be consulted in Table 4.1.

API methods	Description
alloc(s, mem_type)	Allocates Memory with size "s" with different visibilities depending on "mem_type". If "mem_type" is HOST, the memory is only visible by the host. If DEVICE, the memory is only visible by the computing device that allocated it. If SHARED, the memory is visible by all computing devices.
dealloc(ptr, mem_type)	Frees the memory in "ptr" pointer allocated with type "mem_type".
atomicCAS(var, cval, nval)	Atomically tests if the variable "var" has the "cval" value. If successful, "var" is updated with "nval"
atomicAdd(var,val)	Atomically adds the value "val" to the current value of "var"
atomicSub(var,val)	Atomically subtracts the value "val" to the current value of "var"
synchronize()	Acts as a barrier for all the currently executing tasks. After this point all the previously launched tasks are guaranteed to have finished and the data on the host is coherent

Table 4.1: Low level memory API

4.2 Distributed shared memory system

The proposed distributed shared memory system is designed for heterogenous computing systems with CPUs and accelerators. The heterogenous DSM unifies the disjoint address spaces of host main memory and accelerator memory (GPU address space), freeing the programmer from having to explicitly manage data transfers across computing devices.

4.2.1 Conceptual model

The heterogeneous DSM organizes its memory pool into pages with the same size. A page is the smallest unit of transfer between physical address spaces. The

DSM follows a single writer, multiple reader model (SWMR), i.e. only one valid copy of a page with write permission exists on the system at any given time, but several valid copies of read-only pages are allowed. Note that if a page with write permissions exists, then no read-only copies of the same data can exist simultaneously. The system is centralized on the host, which is responsible for serving data requests and guarantees coherence. The memory system is hierarchically organized in two levels – (1) central memory, residing on the host address space, where the original data is stored before task execution and where data requests are fetched from; and (2) device memory, computing devices' private memory where copies of necessary pages will reside during task execution. Page coherence is maintained using the MSI protocol (Modified, Shared, Invalid). A high level diagram of the model and how it works can be viewed in Figure 4.3.

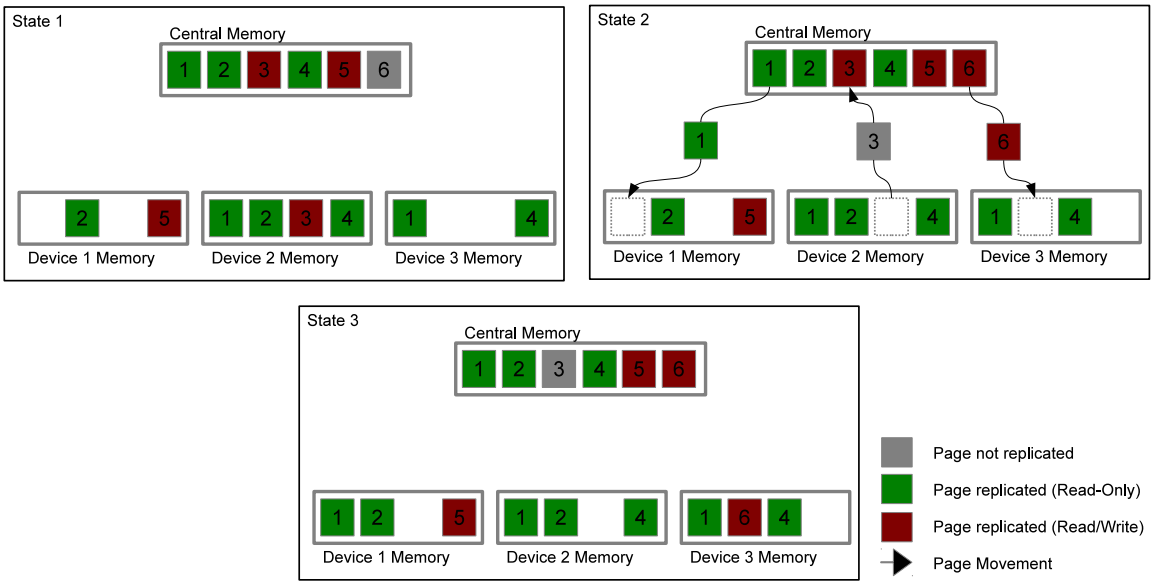


Figure 4.3: Diagram of the DSM mechanism of data replication between central memory and all the computing devices. The diagram shows the Device 1 receiving a Read-Only page from central memory; Device 2 returning a Read/Write page to central memory; and Device 3 receiving a Read/Write page from central memory.

Centralized vs Distributed manager approaches

In regard to data location, two groups of solutions exist: centralized approaches and distributed approaches. A distributed solution scales better but it is also more complex. The distributed approach would require not only the CPU but also the accelerators to deal with all the necessary bookkeeping operations associated with a distributed solution (manage data state, data location, incoming/outgoing request queues, etc.). This solution would require a high cyclomatic complex implementation, to which simple accelerators like GPUs are very sensible to. Also, this type of solution generally requires fine-grain thread control capability, which the GPU does not have as well.

By following a centralized approach on the host, the memory runtime system enables the use of simple accelerators and minimizes the overhead on them. The centralized solution could potentially limit scalability, but given the limited number of acceleratores that can be fitted in a single machine, the scalability impact should be minimal. Efficient techniques of doing the data transfers between the central memory and the devices' memory (discussed further on) can also reduce the problem. In short, the centralized approach introduces the least computational overhead on the accelerator side, enables the use of feature poor accelerators, and does not use features not guaranteed to work in future architectures by manufactures.

Coherence protocol and Consistency model

The memory runtime system uses a set of centralized tables on the host to keep track of all the pages state and location on the system. There is a table for each device that matches current device pages with their state. Since the system uses the MSI memory coherence protocol, each page can be in one of three states: Modified, Shared or Invalid. Modified means that the computing device has a page that was altered and needs to be copied back to the host; Shared means that page is valid and may or may not exist another copy of the page in other device; and Invalid means that the page is outdated and a recent copy must be fetched from the host. All these states are defined from the central memory perspective.

The heterogeneous DSM follows an implicit Acquire-Release consistency model.

When a task is scheduled to run, the runtime memory system determines the task data requirements. If read/write data is required, an acquire action is performed on the corresponding memory pages and the data is moved onto the respective device prior to the task execution (data prefetching). After execution the respective pages are copied back and a release action is performed. All these actions are performed automatically by the runtime system with no programmer intervention. Read-only data does not require the use of the consistency mechanism.

One of the limitations of using heterogeneous systems is the inability to use atomic operations, since these operations only guarantee atomicity to threads running on the same device. The implicit Acquire-Release consistency model only guarantees page exclusivity to a device and not to a thread, i.e. guarantees inter-device exclusivity and not intra-device exclusivity. This further relaxed Acquire-Release model allows the programmer to control resources with atomic operations. The use of efficient, architecture specific and exclusive, atomic operations is thus possible, because the memory system transparently guarantees data exclusivity on the computing device. This way, the use of atomic operations is made viable in an heterogeneous environment. It's important to note that the programmer uses a set of architecture independent atomic operations to write the kernels and the framework replaces them with architecture specific ones at compile time.

4.2.2 Implementation

The memory system emulates a shared memory address space by first reserving memory from main memory (host memory) and accelerators' private memory (device memory). The host memory will account for the totality of memory available to the runtime system i.e. all the data allocated during execution must fit in this pool. The device memory, generally smaller than host memory, serves as an additional intermediate memory between the processor and the main memory. Effectively, the devices' memory can be viewed as an additional level of cache memory. These memory pools are structured in pages of 4KBytes (configurable), matching the operating system's and GPU driver's page size to avoid false sharing and improve performance.

After the scheduler assigns a task to a computing device, before the task

is executed, the runtime memory system is called to make the necessary data movements. The two data methods specified by the programmer, that indicate the data requirements, are called. With these two methods, the memory runtime system should then generate a list of pages with read/write and read-only permissions. This list of pages is then cross-checked with the page tracking table. The test checks three different aspects: (1) if the page has read/write permissions, i.e. if it's free (no lock in effect) to be copied; (2) if the page is already in the device (if so, no copy is necessary); and (3) if the page is not going to evict another page already on the device memory, required by an executing thread. After the test, a new list of pages is outputted (depending on the outcome of the test for each page) and the data is moved to the computing device.

The memory runtime system keeps track of all the executing tasks. After a task finishes its execution, the page tables are updated and the necessary data is copied back from the computing device to host memory. Only pages marked as read/write are copied back.

Data movement optimizations

Since the overhead of moving data across PCI-Express is significant, doing a single copy for each page is not sustainable. To reduce the cost of moving the necessary data to device memory, the page copies are coalesced into the minimum number of memory copies possible. To do this, the list of pages is ordered in regard to their physical memory positions and then transferred in bulks of contiguous memory positions. The process is represented in Figure 4.4. After the data is scheduled to move, the task can be executed.

If the computing device does not have dedicated memory and shares the host address space, no data copies are performed. The necessary acquire and release operations and the additional bookkeeping operations to guarantee memory coherence are still done, but no actual data movement is performed, since the host memory is the device's private memory as well. These devices are said to have a virtual private memory. In the current version of *GAMA*, the CPU itself fits the profile of a dedicated memory-less device. Despite being the processor responsible for managing the system, it is also capable of doing effective work and in

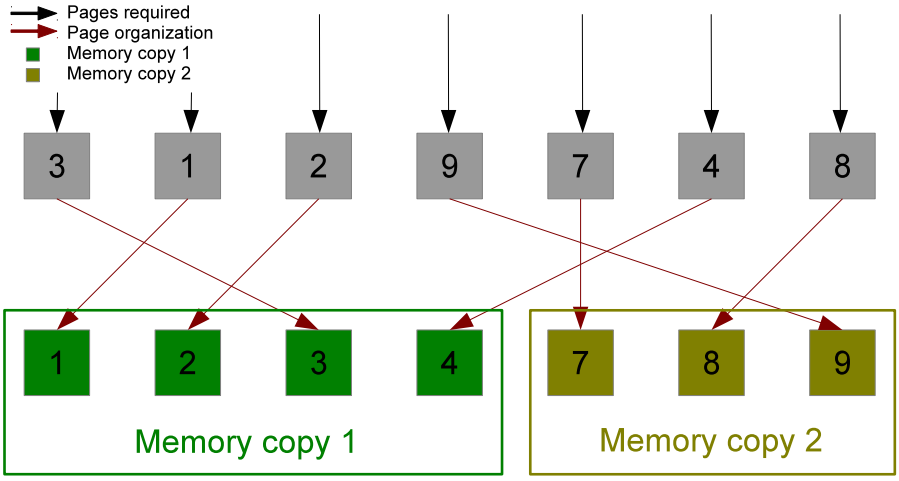


Figure 4.4: Diagram of the of a coalescing process of seven pages into two memory data movement operations

this context is a computing device like the accelerators. Since the CPU dedicated memory is the host memory (by definition), as a computing device, it is seen as any other accelerator but with no dedicated memory.

Modern accelerators can be set to use host memory instead of their dedicated memory to simplify programming effort and accommodate large problem sizes at the cost of performance. The heterogeneous DSM is able to take advantage of this capability. If the hardware supports it, the caching mechanism can work in hybrid mode by having some data in accelerator dedicated memory and the rest in host memory. This process is transparent to the programmer and if the data required by a given task is larger in size than the dedicated memory, the memory runtime system will copy what it can to the dedicated memory (caching) and access the rest through DMA. Although not ideal, since access times to different memory positions might be considerably different, it makes the system more flexible.

Address translation and Replacement strategy

When shared data is accessed, the respective global virtual addresses are translated into device physical memory at access time. The translation mechanism is hidden from the programmer through the use of smart pointers. These pointers

have their access methods overloaded with methods that translate the addresses during the access.

To reduce the overhead of address translation, the mapping mechanism from virtual addresses to physical ones, follows a direct mapping approach. An alternative would be the use of a n-way associative mapping policy to reduce conflict misses, but this comes with a higher address translation time cost. Although the direct mapping mechanism is prone to trashing, it provides a way to do address translation in constant time with little overhead.

The trashing problem is also more prone to happen in small caches (several KBytes/few MBytes), since close addresses might overlap if the current working domain is larger than the cache (not difficult in small caches). But since accelerators dedicated memory, like a GPU, tends to vary between several hundreds of MBytes to few GBytes the trashing problem is somewhat masked.

Runtime optimizations

The memory runtime system uses several techniques to reduce memory transference latency – memory transfers coalescing, asynchronous data copies and data prefetching. The last is achieved by cooperating with the runtime scheduler. By performing the data transfers asynchronously, the memory runtime system is able to move the data to the devices while they are executing tasks, thus reducing data transfers penalties. However, the runtime memory system needs to know what and where the data will be required.

Before a running task is over, the runtime scheduler can signal the memory runtime system as to which will be the next task(s) to execute and where. The runtime memory system can then perform the necessary operations (evaluate the data requirements, check what pages need to be copied and coalesce the memory copy operations) and copy the required data to the respective device while the device itself is performing actual work. Since these operations are done asynchronously, they can potentially be over before the current executing task of the device is done executing and the new task (that requires the new data) starts its execution. This minimizes idle time between execution of different tasks and data movement penalties.

4.3 Dynamic memory manager

The heterogeneous memory system is fitted with a dynamic memory allocator. This memory allocator was designed to accommodate high thread volume environments running on different architecture parallel processors simultaneously.

4.3.1 Conceptual model

The memory manager maintains as many heaps as processing cores plus a shared pool (P heaps + 1 shared pool). A running thread can only access its respective heap and the shared pool. The memory is organized into three groups hierarchically – blocks, super-blocks and hyper-blocks. A block is the minimum size unit that a thread can reserve (small size requests will be round up to a block) and a super-block is a group of 64 blocks. When the allocator receives an alloc request, it calculates the number of blocks necessary to satisfy the request. If the cores private heap does not have the number of blocks requested available sequentially, a new super-block is fetched from the shared pool. A super-block is the minimum amount of memory space that can be fetched from the shared pool. Requesting several blocks at one time helps reduce the number of times a thread needs to access the shared pool and thus reduces the serialization points in the system.

If the memory request size does not fit in a super-block, a hyper-block is fetched from the shared pool to satisfy the request. A hyper-block is a group of super-blocks that is used for large memory request sizes (requests bigger than a super-block). A hyper-block contains the minimum amount of super blocks necessary to satisfy the request. After receiving a deallocation request, if the respective heap manages to get one or more (in case of a hyper-block) super-blocks completely free, those super-blocks are returned to the shared pool. This recycling mechanism assures that available space is returned to all threads of the system as soon as possible and reduces fragmentation.

The shared pool stores two types of data and can be divided into two pools: free super-blocks memory pool; and free hyper-blocks memory pool. Both these pools are kept in a LIFO structure to improve locality. When a memory position is freed and the heap must return free space to the shared pool, in case of a freed super-block, it will be returned to the free super-blocks memory pool and

in case of a hyper-block, it will be stored in the free hyper-blocks memory pool. When a new allocation is made, if a private heap needs a new super-block, it will search for free super-blocks first in the free super block memory pool, and if a new hyper-block is needed, the allocator will first search for free space in the free hyper-blocks pool. This improves the allocation speed of hyper-blocks. Searching the free super-blocks pool to satisfy a hyper-block request can be slower since the free super-blocks pool does not necessarily have the free super-blocks sequentially organized. This requires the allocator to check for sequential groups of super-blocks to form a new hyper-block. By storing previously allocated hyper-blocks on an specific pool, the effort of creating a new hyper-block out of free super-blocks does not have to be repeated. A high level diagram of this model can be viewed in Figure 4.5.

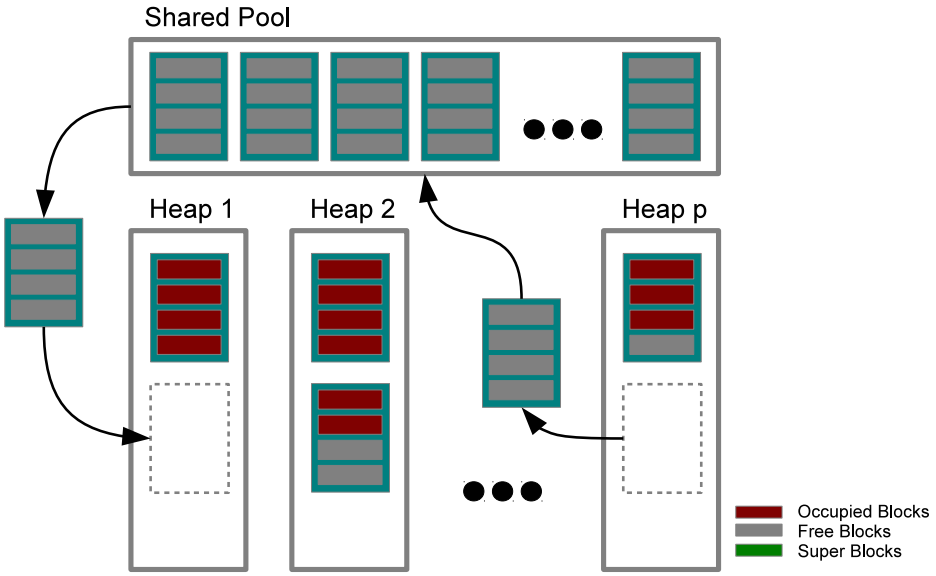


Figure 4.5: Diagram of the memory allocator showing one heap returning a free super-block to the shared pool and another heap fetching a new super-block from the shared pool.

4.3.2 Implementation

The allocator uses a hash function to map each individual thread with its corresponding private heap. This process can result in collisions, but given the *GAMA* execution model, that assigns the same number of worker threads (responsible for the execution of tasks) as processing cores, collisions can always be avoided. Each private heap is implemented as a double linked list of super-blocks as are the free super-blocks and free hyper-blocks memory pools. To keep track of the state of the super-blocks, a 64 bit mask, for each super-block, is stored and identifies the free and occupied blocks of the super-block. Together with the mask, an additional flag identifies the super-block as being part of an hyper-block or not.

The 64 bit masks, despite being used to identify the free and occupied blocks locations in a super-block, can also, incidentally, be used to know the number of free blocks in a super-block. However, using the masks to find the number of free blocks requires somewhat complex bitwise arithmetic that without hardware support cannot be done in constant time and can be time consuming. Since neither the CPU or the GPU have special units to count the number of true or false bits in an word/long word, an additional structure is used to store the number of free blocks in a super-block. Despite storing redundant information, this helps speed up the search for available space on the super-blocks of a core's heap. In the case of a super-block being part of an hyper-block, this structure stores the number of super-blocks in that particular hyper-block.

Despite having multiple double linked lists that can vary in size, they were implemented without the use of dynamic memory. This is possible given two characteristics of the proposed model: the total number of super-blocks in the system is constant and each super-block can only be reserved in a single private heap or be available in the free super-blocks or hyper-blocks pool at any given time. To implement the double linked lists two static arrays are used: the *next_sblock* and the *prev_sblock*. Each index of the arrays represents a super-block and the value stored in the arrays in that index, represents the index of the next and the previous super-block (stored in the *next_sblock* and the *prev_sblock* arrays respectively). A negative index represents a null connection. Each new list in

the system only needs to store the index of the first element and use these two arrays to find the other super-blocks of the list. Since the number of super-blocks is constant, the size of the arrays is also constant and since each super-block can only belong to a single list at any given time, only these two arrays are required to keep track of all the lists in the system. This enables the use of flexible structures like double linked lists without requiring the use of dynamic memory, which improves data locality and avoids dynamic memory allocation and deallocation overhead.

Processing core definition

Since the allocator assigns a private heap to each core in the system, it is important to define a processing core in the heterogeneous memory allocator context, since different manufacturers have different definitions from each other and even among themselves when referring to different architectures. In this context, a processing core is an independent execution unit capable of decoding and execution of a thread in SISD or in SIMD fashion. With this definition a CPU core is the same as defined by the manufactures, a GPU core however is not. GPU manufacturers qualify a core as a single unit capable of executing an arithmetic operation. However, these units are packed in groups since they share the same fetching and decoding unit, the same instruction pointer, and some of these arithmetic units can only do special types of arithmetic operations. In this context, we call a GPU core as an independent group of these simple computing units and qualify it as an SIMD "number of simple units per group"-wide processing core. Using manufacturer nomenclature, a core is an SM in Nvidia GPUs (Fermi architectures) and a compute unit in AMD GPUs (GCN architecture[24]), 32 and 64 wide respectively.

Block and super-block sizes

Although configurable, the block size should be a multiple of the DSM page size. Like the DSM itself, that uses the same page size as the operating system and the GPU driver to avoid false sharing, the memory allocator (that runs on top of the DSM) needs to follow the same methodology, for block sizes, for the

same reasons. Since the minimum amount of memory that can be reserved by any thread is a block, no two threads will require the same memory page to access data allocated in two different blocks, i.e. there is no possibility of false sharing and no unnecessary page bouncing between two different devices (ping-pong effect).

The choice for a 64 block sized super-block was both a compromise and a choice of convenience. Since the super-block is the minimum size of free memory that can be fetched and returned to the shared memory pool, and being the shared memory pool the only serialization point between independent processing cores, the super-block size is relevant for the memory allocator performance. A smaller block size reduces fragmentation, but increases the chance that a heap might need to access the shared pool and the chance that it might have to wait to access it. A bigger block-size reduces the chances of accessing the shared pool and thus serialization chances, but increases the memory fragmentation. Preliminary tests showed the 64 block sized super-block as a good compromise between serialization chances and fragmentation in terms of performance. The performance results are also influenced by the fact that the development and test machine used 64 bit processors. The mask arithmetic, necessary to manage the super-blocks state, didn't perform faster with smaller super-block sizes (e.g. 32 block size with 32 bit masks) but performed slower with bigger super-block sizes (e.g. 128).

Almost lock free approach

The allocation and deallocation follows an almost exclusive lock free approach. This is achieved by using compare and swap (CAS) operations. When a thread approaches a critical section, it reads a value and tries to updated with an atomic CAS. If successful, it will enter the critical section, if not, meaning the value was altered by another thread, the operation is repeated until successful. This is the methodology followed by several lock free scalable parallel memory allocators.

The lock free (atomic operations only) approach is problematic when dealing with heterogeneous systems. The atomicity of these operations is only guaranteed among intra device threads. Atomic operations executed by two threads running in different processors (e.g CPU and GPU) are not atomic. This is not generally a problem since each processor has its private heap, but the shared heap is accessible

by all devices. When accessing the shared heap the atomic CAS approach is not viable and distributed locks are necessary. The implemented solution is a modified version of the Eisenberg and McGuire algorithm[14]. This algorithm allows each thread to enter the critical section only when they have the token. The token is passed around and always belongs to one and only one thread at a time. Before receiving the token, the threads should first announce their intention to enter the critical section. As soon as their turn comes they receive the token and can enter the critical section. At the end, the token is passed to another thread waiting for its turn. In an effort to minimize the performance impact of this lock, in an otherwise lock free implementation, an hybrid Eisenberg and McGuire and atomic CAS algorithm was implemented. Instead of alternating the token between all threads on the system, the token is used only to select computing devices. Once a device has the token, the atomic CAS approach is used to control the critical section between the device's threads.

GPU specific optimizations

One of the shortcomings of the GPU is that, even though it allows thread diversion in the same thread warp, threads running simultaneously on the same SM, have to share the same instruction pointer. This is achieved by the hardware by grouping the threads that share the same execution path and running the different groups sequentially. This is an hybrid between MIMD and SIMD models that Nvidia in particular calls SIMT (Single Instruction Multiple Thread). When GPU threads try to allocate memory, all threads running in the same warp will reserve different memory positions which translates into a unique execution path for all threads in a warp. This represents the worst case scenario in an SIMT processor since all threads will run sequentially.

To avoid this issue, GPU threads allocate memory in a different way than the CPU threads. When the threads in a warp try to reserve memory, only one will be able to actually place the request. However, this thread (chosen randomly) will request the necessary data for all threads in its warp. Since intra block threads are able to synchronize and comunicate quickly, all threads can register their request sizes and a single thread can place the actual reserve request. After the request is

satisfied, the thread that allocated the memory can comunicate resulting memory pointers to the other threads. This communication and synchronization is done through on-chip fast memory, which results on a penalty of only a few clock cycles. This effectively removes the serialization issue and speeds the allocation time in the GPU.

Chapter 5

Validation

> *A fact is a simple statement that everyone believes. It is innocent, unless found guilty. A hypothesis is a novel suggestion that no one wants to believe. It is guilty, until found effective.*
>
> – *Edward Teller*

In this chapter the proposed heterogeneous distributed memory system is validated on the *GAMA* framework. It starts with a description of the languages and tools used for development and tests. It continues with a brief description of the test cases used to benchmark the DSM and ends with the presentation and discussion of the results obtained. The tests made aim to compare the *GAMA* framework with and without the proposed DSM in different aspects such as performance and scalability.

5.1 Experimental setup

All the tests in this document were made in the same heterogeneous machine with multiple CPUs and GPUs. The machine is composed of two Intel Xeon processors with 6 cores each and three Nvidia's GTX 580 GPUs. The machine was running the Linux operating system with 2.6.43 kernel. The exact details of the machine can be seen in Table 5.1 and both Intel and Nvidia's processors details can be consulted in Table 5.2.

Machine components	Details
CPUs	2 Intel Xeon E5645
GPUs	3 Nvidia GTX 580 (1.5GB)
Main memory	12GB (ddr3 1333MHz)
Operating System	Fedora linux (love lock)
Linux kernel	2.6.43.8 (x86_64)

Table 5.1: Test machine details

Device	CPU	GPU
Name	Xeon E5645	GTX 580
Architecture	Nehalem	Fermi
Code Name	Westmere-EP	GF110
Core clock	2.4GHz	800MHz
Shader clock	–	1600MHz
Cores	6	512(32*16SMs)
L1 cache	32KB ic + 32KB dc per core	64KB per SM
L2 cache	256KB per core	768KB shared
L3 cache	12MB shared	–
Year	2010	2010

Table 5.2: Test machine's processors details

Since currently the *GAMA* framework is being developed in C/C++ and in C for CUDA, the proposed heterogeneous distributed shared memory system and memory allocator was also developed in C/C++ and C for CUDA. The C/C++ portion of the code was compiled with GCC 4.6 and the C for CUDA portion was compiled with NVCC 5.0 (release candidate), with -O2 optimizations in both compilers.

5.2 Case studies

Aside from the synthetic tests used to evaluate particular aspects of the proposed heterogeneous memory system, three additional test cases were implemented in *GAMA* and used to benchmark the framework with and without the proposed memory system. The test cases used are SAXPY, and two n-body simulation algorithms: a naive direct sum algorithm[31] and a Barnes-Hut algorithm[5].

5.2.1 SAXPY

SAXPY stands for Single-precision real Alpha times X Plus Y and is a first level (vector-vector) operation in the Basic Linear Algebra Subprograms (BLAS) package. The operation computes $Y \leftarrow \alpha X + Y$, where X and Y are both vectors and α is a scalar value.

The algorithm is easily parallelized since each element of the result vector Y can be computed independently, i.e. for a vector size n, each element Y_i with $1 <= i <= n$, can be calculated as $Y_i = \alpha * X_i + Y_i$. This algorithm has the advantage of exposing a lot of parallelism and can be easily coded, however it has a low ratio between computation and memory accesses making it memory bound. The SAXPY algorithm is an excellent test case since it is a very used BLAS operation.

5.2.2 N-body simulation (all-pairs)

N-body simulation is the simulation of the movement of particles (bodies) in space under the influence of forces (gravitational, electric, magnetic, etc.) from other bodies in the system. The n-body problem was originally proposed by Isaac Newton and proved that a spherical body can be modelled as a point mass. The n-body problem has been completely solved for 2 bodies but is deemed impossible for an arbitrary (larger than 2) number of bodies. Numerical integration methods exist for these cases, but the solutions are an approximation.

The all-pairs method evaluates the resulting net force applied on a body by all other bodies pair-wise. This process is repeated for all the bodies in the system. Although simple, this method has a computational complexity of $O(n^2)$ which makes it prohibitively slow for a large input set. Given the parallelism potential of the algorithm, implementations of it in many-core architectures have produced good results[31].

This algorithm, much like SAXPY, exposes a lot of parallelism but also has the characteristic of being cache friendly by having significant data reutilization. This is the kind of algorithm that the GPU handles well and should provide a good test to the software cache of the DSM when compared with the existing hardware cache mechanism.

The algorithm was implemented in *GAMA* following a similar approach developed by Nyland et. al.[31]. Although each pair interaction between bodies can be done in parallel, such a fine thread granularity would require a lot of memory and produce a memory bottleneck, since the results of each pair-wise interaction would have to be stored so that the resulting net force could be later calculated. The implemented algorithm reduces the parallelism by sequentializing the computation of the resulting net force for a single body and doing this operation in parallel for each body. This reduces the available parallelism from $O(n^2)$ to $O(n)$.

5.2.3 N-Body simulation (Barnes-Hut)

The Barnes-Hut algorithm[5] is used for N-Body simulation as an alternative to the brute-force all-pairs method. The algorithm recursively structures the bodies hierarchically according to their distance from each other by grouping them into increasingly smaller boxes/cells. This structure is stored in an octree (a three dimensional binary tree), where each of the eight branches represents a 3D octant and further branches of each octant represent a similar subdivision of the respective octant. This subdivision stops when all bodies have a private octant (no other bodies in the cell). Information about each octant is also stored such as the mass and center of mass of the bodies it contains. An example of this division in 2D space can be viewed in Figure 5.1.

To calculate the net force between bodies, for each body, the octree is traversed and the interaction between other bodies is calculated. If a body (or group of bodies) is deemed far enough, the respective mass and center of mass of that octant is used instead of each individual body in the octant. A body is determined to be far enough away if the ratio between the width(s) of the octant the body is in and the distance between the body and the octant's center of mass(d) is bellow a threshold (θ) i.e. $s/d < \theta$. The threshold is commonly set at 0.5 ($\theta = 0.5$). This approximation reduces the algorithm complexity from $O(n^2)$ to $O(n\,log\,n)$.

Since this is a tree based algorithm, the Barnes-Hut algorithm follows an irregular memory access pattern. This is an issue for automatic caching mechanisms since these algorithms do not explore data spacial locality causing a lot of cache misses. This algorithm presents itself as a difficult class of algorithms for

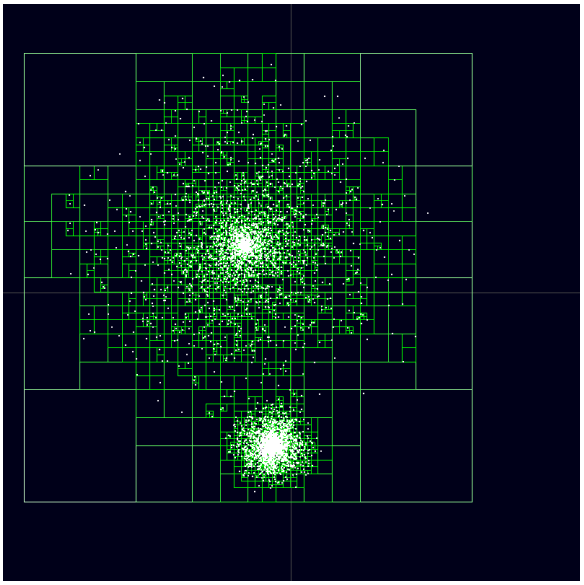


Figure 5.1: Complete Barnes-Hut tree decomposition of a distribution of 5000 particles in 2D space (source Wikimedia).

automatic caches and provides a challenge for both the proposed DSM's software prefetching and coalescing mechanism and the GPU hardware's.

The algorithm was implemented in *GAMA* and was influenced by Burtscher's et. al.[10] implementation. Burtscher et. al. proved the suitability of irregular code on many-core architectures by implementing and optimizing the Barnes-Hut algorithm on an Nvidia's CUDA GPU and achieving results 74 times better than an optimized sequential CPU version. Despite this implementation being GPU tailored, a parallel CPU version was also developed to test the suitability of the approached used, but optimized for parallel CPUs. The code was implemented in OpenMP and achieved better results than all the openly available parallel Barnes-Hut CPU implementations.

The version implemented in *GAMA* takes advantage of the framework flexibility and uses a GPU tailored code (similar to Burtscher's) when running a task on a GPU and a CPU tailored version (based on the OpenMP version) when the task is ran on a CPU. It is important to note that this distinction and the architecture specific binary switch is made at runtime by the framework and is done automatically.

5.3 Results

The results are organized into two categories: the proposed distributed shared memory system and the parallel memory allocator. The parallel memory allocator was tested with synthetic benchmarks and the DSM was tested with the three algoritms presented previously. All reported values represent an average of 30 test runs.

5.3.1 DSM performance results

The first set of tests was made using the SAXPY algorithm. The results when using a single CPU and a single GPU can be viewed in Figure 5.2. For every input size used, the algorithm is 30% slower on average when using the DSM. This result can be justified by the lack of data reutilization and the low ratio between computation and memory accesses in the algorithm. The computation done by each individual thread requires three memory accesses, two loads and a store, all done in different memory locations. Since there is no data reutilization, the data cached in the device's memory by the DSM does not speeds up memory accesses, actually it slows the algorithm down since it introduces an additional memory copy.

The performance impact is aggravated further by the low ratio between computation and memory accesses. Each thread only has to do two arithmetic operations for each three memory accesses. This is a problem since the DSM has to translate virtual memory addresses into physical addresses in each memory access. Since there are more memory accesses than computation on the algorithm, the DSM introduces more computation overhead, due to the address translation, than the effective computation present on the algorithm.

The regular memory access pattern of the algorithm also has some impact in the DSM performance with this algorithm. The GPU is designed to hide memory access latency using fast thread switching between threads that are waiting for data with ones with the data ready, and coalescing independent memory requests with a single one reducing request contention on the bus. These mechanisms work exceptionally well on algorithms such as SAXPY. This translates into an algorithm that having the data in host memory, instead of device's memory,

causes little performance impact. All these characteristics of the algorithm and the DSM add up, causing a bad performance of the framework using the DSM when compared with the non DSM version.

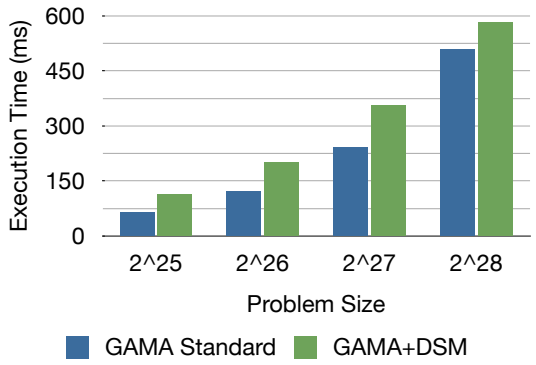


Figure 5.2: SAXPY algorithm results running on a single CPU/single GPU configuration, with and without DSM for for different problem sizes (from 2^{25} to 2^{28} elements)

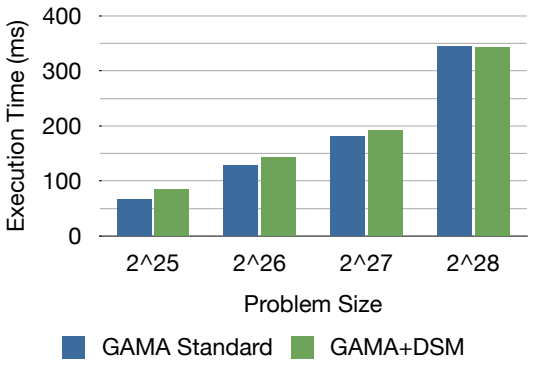


Figure 5.3: SAXPY algorithm results running on a single CPU/two GPU configuration, with and without DSM for for different problem sizes (from 2^{25} to 2^{28} elements)

Despite the bad results of the DSM for the SAXPY algorithm using a single CPU/single GPU configuration, when using a single CPU/ two GPU configuration, the results are slightly more encouraging. The results for this configuration can be viewed in Figure 5.3. The DSM version in this case is only about 10% slower on average than the no DSM version. This is due to the better scalability of the framework for this algorithm when using the DSM. The two GPUs version is only 1.2 times faster on average than the single GPU version, but the two GPUs version with the DSM is 1.6 times faster on average than its single GPU version counterpart. The use of multiple GPUs introduces contention problems on the PCI-Express bus. This can be a scalability bottleneck if the bus is saturated which means that, despite the non DSM favorable nature of the algorithm, the DSM version can be faster than the non DSM version for a large enough input set as verified by the results for an input size of 2^{28} elements.

The next algorithm tested was the all-pairs N-Body simulation algorithm. The results of the algorithm running on a single CPU/single GPU configuration for multiple problem sizes can be viewed in Figure 5.4. The results show that

the DSM version performs slower than the non DSM version for small input sizes (smaller than 2^{19} bodies), and performs significantly faster for larger input sizes (bigger than 2^{20} bodies)

This algorithm does not have the same low ratio between computation and memory accesses problem as the SAXPY algorithm, but, like the SAXPY algorithm, it has a regular memory access pattern. Not only this N-Body simulation algorithm has a regular access pattern, but also has significant data reutilization. Both these characteristics allows for a good performance of the hardware cache (the algorithm explores both temporal and spacial locality) and a significantly worse DSM performance, due to the additional overhead, for a small input sizes.

However, the DSM version is faster than the standard version for the 2^{20} and 2^{21} bodies input sizes – 1.6 times faster. Since the algorithm is cache friendly, it should perform well with the DSM version of *GAMA*, however, with the small problem input sizes that doesn't happen. The reason is that the problem input sizes are not big enough to hide the DSM overhead. But with a big enough input size such as 2^{20} bodies, the overhead introduced by the DSM is masked by the effective work and the algorithm is able to take advantage of the device's faster dedicated memory.

The results for the single CPU/two GPU configuration, shown in Figure 5.5, follow the same pattern as the single CPU/single GPU configuration discussed above – the DSM version performs faster than the standard version for big input problem sizes. Note that, since the the algorithm has a high ratio between computation and memory accesses, the framework does not have the same scalability problems as with the SAXPY algorithm. With the 2^{20} and 2^{21} bodies input sizes, both two GPUs configurations with and without DSM are 2 times faster than their single GPU version counterpart. This supports that the DSM version is faster than the standard version for large input sizes and slower on smaller input sizes because of the overhead introduced by the DSM and not by PCI-Express saturation for large input sizes like the SAXPY algorithm.

The last algorithm tested was the Barnes-Hut N-Body simulation algorithm. The results of the algorithm running on a single CPU/single GPU configuration for multiple problem sizes can be viewed in Figure 5.6. The DSM version is 4.4 times on average better than the non DSM version. This algorithm differs

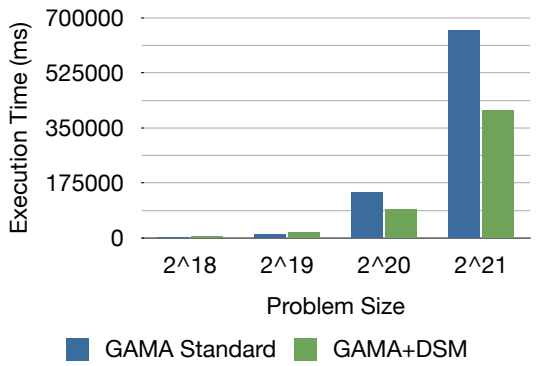


Figure 5.4: N-Body (all-pairs) algorithm results running on a single CPU/single GPU configuration, with and without DSM for for different problem sizes (from 2^{17} to 2^{20} bodies)

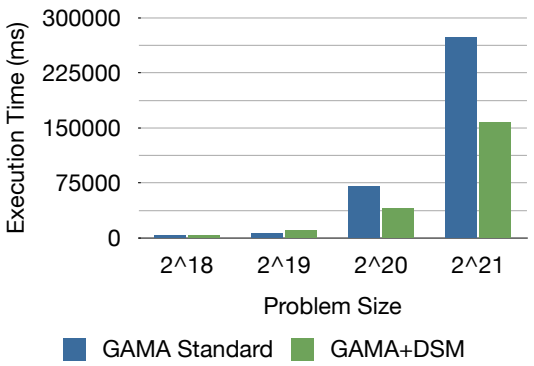


Figure 5.5: N-Body (all-pairs) algorithm results running on a single CPU/two GPU configuration, with and without DSM for for different problem sizes (from 2^{17} to 2^{20} bodies)

from the previous ones mainly due to the irregular memory access patterns. The algorithm implementation used in these tests was made considering the GPU drawbacks and limitations and considerable work was done to reduce the impact of these limitations. Despite the effort done, such as reducing the memory access irregularity, the algorithm is still irregular. The irregularity translates into poor hardware memory coalescing and high cache miss ratio, i.e., many main memory accesses. The DSM version reduces the issue since it moves the data to a closer (faster) memory for the device(s) (from host memory to device's memory). This reduces hardware cache misses latency penalties and significantly improves the algorithm's performance.

The results of the algorithm running on a single CPU/two GPU configuration for the same problem sizes can be viewed in Figure 5.7. The DSM version is 7 times on average faster than the non DSM counterpart. This improvement of the two GPU configuration versus the single GPU configuration is due to both good scalability of the DSM version and poor scalability results of the framework without DSM. The DSM with two GPUs version is 1.85 times faster than single GPU version on average, and up to 2.2 times faster in one case. On the other hand, the no DSM two GPU version is only 1.15 times faster than the single GPU configuration version on average. As the DSM results with two GPUs for other

algorithms presented previously, the results for this algorithm show that the DSM improves or maintains the advantage of the non DSM version when compared the sigle GPU configuration results.

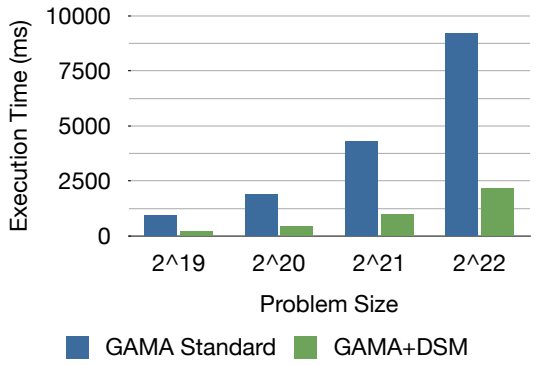


Figure 5.6: N-Body (Barnes-Hut) algorithm results running on a single CPU/single GPU configuration, with and without DSM for for different problem sizes (from 2^{19} to 2^{22} bodies)

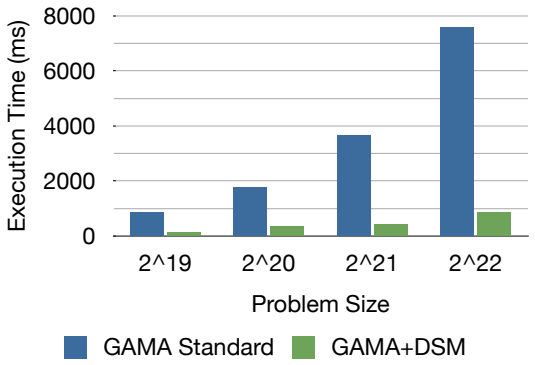


Figure 5.7: N-Body (Barnes-Hut) algorithm results running on a single CPU/two GPU configuration, with and without DSM for for different problem sizes (from 2^{19} to 2^{22} bodies)

Since the DSM acts as an additional cache level, one would expect it to suffer from the same problems as the hardware cache when faced with an irregular algorithm, i.e., many cache misses. The DSM improves on the hardware caching mechanism by (1) having a larger memory size available for caching and (2) by having a more efficient prefetching and coalescing mechanism. The advantage of having a larger cache is self explanatory, the bigger the space available to store the problem data the smaller is the probability of the data not being in cache. In this particular case the cache size improves more than 1000 times, from several hundreds of KBytes to a few GBytes. The second factor is the prefetching and coalescing mechanism. Since the DSM allows the programmer to specify the data requirements of the algorithms as a method, the mapping between threads and their data requirements can be as complex and specific as the programmer wants it to be. Using this "thread-data mapping" method, the prefetching and coalescing mechanism can be tuned for the specified algorithm.

Overall the DSM has acceptable performances on non favorable algorithms like the SAXPY algorithm by allowing multi GPU scalability, and great results

Algorithm	SAXPY	N-Body (All-Pairs)	N-Body (Barnes-Hut)
CPU + 1 GPU(s)	0.874	1.614	4.265
CPU + 2 GPU(s)	1.009	1.730	8.694

Table 5.3: Speed up of all algorithms when using the DSM vs not using, for the different accelerators configurations. The speedups were calculated using the biggest problem size for each algorithm, i.e. 2^{28}, 2^{21} and 2^{22} for the SAXPY, N-Body (All-Pairs) and N-Body (Barnes-Hut) respectively.

on irregular algorithms like the Barnes-Hut N-Body simulation algorithm by improving on the non DSM version significantly (both on single and multiple GPUs configurations). The DSM also significantly improved the performance on regular algorithms with data reutilization, but only for large enough input sizes. The SAXPY algorithm had the worse results but given the nature of the algorithm itself, the results are somewhat expected. The DSM results with two GPUs showed improved performance when compared to the respective non DSM versions. Even the "non DSM suitable" SAXPY algorithm showed better results by, not only closing the performance gap between the DSM and non DSM versions, but by exceeding the non DSM version for a large problem size. The all-pairs N-Body simulation algorithm saw a negative impact in performance with the DSM for small input sizes, but for large enough input sizes, the DSM was able to mask the overhead introduced and the algorithm was able to take advantage of the faster memory provided by the DSM. Some speedups of the DSM version vs the non DSM version can be viewed in Table 5.3.

5.3.2 Memory allocator results

Even though none of the algorithms implemented required dynamic memory, the framework itself requires it, e.g. to manage work queues. Since many of these dynamic control structures need to be accessible by all the system's computing devices, the use of a heterogeneous memory allocator is required.

To test the heterogeneous memory allocator, a synthetic benchmark was used to evaluate the allocator performance. The test allocates random sizes of memory in parallel on a 12 core processor (up to 24 simultaneous threads with hyper-threading). Each thread makes several allocations and registers the time. The

result of the test is an average of the allocation time of all memory allocations made by all threads. Figure 5.8 shows the results when using the proposed memory allocator versus the GNU's standard C library allocator for different number of parallel threads. The results clearly show the scalability problems with the standard dynamic memory allocator and point to constant allocation time for the heterogenous memory allocator. Figure 5.9 shows the heterogeneous memory allocator results when compared with the allocation time of the GNU memory allocator for one thread. Even when using 24 threads the heterogeneous allocator is faster than the GNU allocator with a single thread. On average the proposed allocator is 9.8 times faster than the GNU memory allocator.

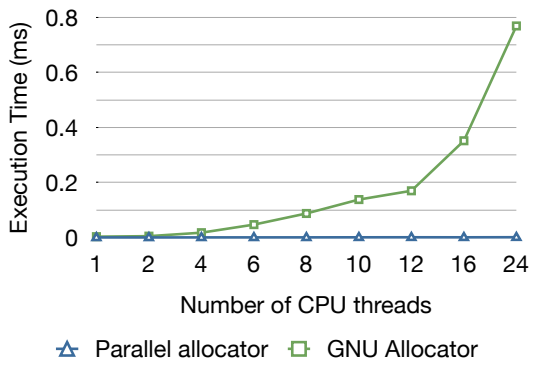


Figure 5.8: Scalability results of the proposed memory allocator and the GNU malloc.

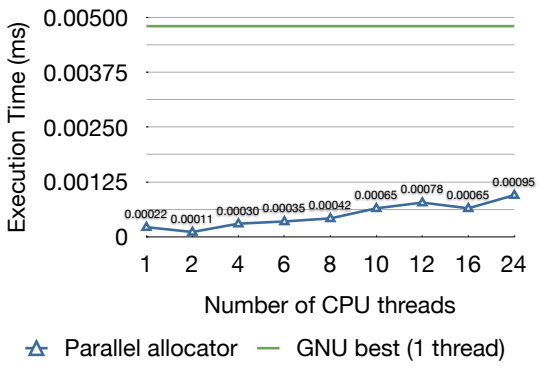


Figure 5.9: Scalability results for the proposed memory allocator vs the best result achieved with the GNU allocator

The memory allocator results running on multiple GPUs configurations can be viewed in Figure 5.10. As can be seen the performance of the allocator on the GPU is much worse than on the CPU. Despite not showing scalability issues when used with two or three GPUs compared to a single GPU, the performance is several times slower than the CPU only configuration. The performance is similar to the GNU memory allocator using 16 threads. The proposed model does in fact assures good scalability performance but the centralized model of DSM has a disastrous impact in the allocator performance of the GPU. Every allocation must check if there is free memory available and what location(s) is(are) available to satisfy the request. Since all the memory is centralized on the host, all these checks must be done in host memory and in the case of the GPUs, must

go through the high latency PCI-Express bus.

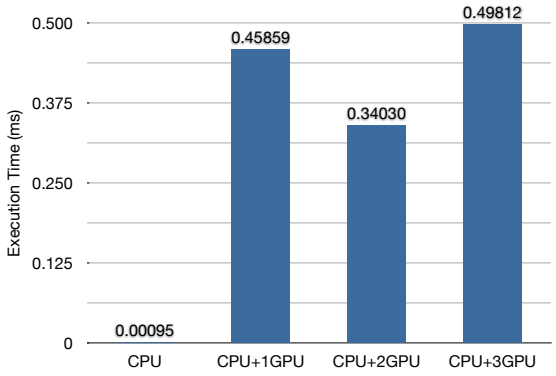


Figure 5.10: Scalability results for the proposed memory allocator when using GPUs together with CPU

5.3.3 Concluding remarks

The proposed DSM system showed to positively impact performance whenever there is data reutilization (temporal locality on memory accesses) and for irregular addressing patterns (where the hardware memory accesses coalescing mechanism cannot improve performance). Additionally, results get better as the data set size increases. This was made evident with the all-pairs code, where the DSM allowed for better results. The DSM worsened SAXPY algorithm performance due to the overhead introduced. The DSM was also able to improve scalability on algorithms with low ratio of computation over memory accesses (SAXPY) and irregular algorithms (Barnes-Hut).

The dynamic memory allocator proved to be more efficient and scalable that the GNU counterpart when only memory sharing CPU cores are involved; whenever there are GPU threads allocating memory, requests must go through the PCI-Express bus compromising performance.

Chapter 6

Conclusion

The most exciting phrase to hear in science, the one that heralds new discoveries, is not 'Eureka!' but 'That's funny...'.

– Isaac Asimov

In this chapter, a summary of the project's contributions is made together with a critical analysis of the project goals and results. The chapter ends with some future work recommendations.

6.1 Project aftermath

This thesis was set on the goal of easing the programming effort of heterogenous CPUs and GPUs systems because of disjoint memory spaces and different memory models, with different consistency models and synchronization primitives. The proposed strategy was to design and implement an heterogeneous memory system that unified the disjoint address spaces, of host main memory and accelerator memory and, allowed the use and allocation of dynamic memory by all computing devices. This system should (1) improve code portability, (2) allow the programmer to be data location agnostic, (3) allow runtime memory management optimizations and (4) allow the use of dynamic memory.

- **Improved code portability** – The proposed memory system follows a single memory model and exposes a single memory API regardless of the

memory models and memory primitives of each of the system's processors/accelerators. This way the programmer does not have to use architecture specific memory primitives (data movement, synchronization, atomics, etc.) or change the algorithm implementation to accommodate different memory models. The generated code is thus agnostic to the system's processors architectures.

- **Data location agnostic programmer** – The DSM organizes the different memory address spaces hierarchically, where the host memory is the main memory (just as in a traditional CPU) and the device's private memory works as an intermediate memory between the device's hardware cache and the host memory. This effectively extends the device's memory hierarchy in one additional level of cache. The device's private memory is managed automatically by the runtime system and, like most traditional caches, is completely transparent to the programmer. The mechanism was designed to support accelerators with different dedicated memory sizes and even with no dedicated memory at all (the accelerator shares the main memory with the CPU), without changing the framework's memory and programming model. This allows the programmer to take advantage of any accelerator's dedicated memory without having to add additional specific functionalities to the code for that effect, or even having to know which (if any) accelerator has dedicated memory – as far as the programmer is concerned, all data resides in a single shared memory pool.

- **Runtime memory management optimizations** – By relieving the programmer from having to manage data movements, the runtime system is then responsible for it. This puts the memory runtime system in a good position to adapt itself to the different families of accelerators and cooperate with the *GAMA* runtime scheduler in an attempt to reduce latency and improve performance. The implemented DSM uses several techniques to reduce memory transference latency – memory transfers coalescing, asynchronous data copies and data prefetching. The coalescing mechanism tries to group several data requests into groups and move the data in each group in a single operation. This helps reduce data movement overhead by min-

imizing the number of transfers between devices. The asynchronous data movement is an excellent way to hide high latency memory copies (e.g. through PCI-Express) with effective computation. By using asynchronous copies the device does not have to wait for the data copy operation to end to start computation. This technique can be paired with the prefetching mechanism. Before a running task is over, the runtime scheduler can signal the memory runtime system as to which will be the next task to execute. With this information the memory system can evaluate the data dependencies of the task, determine which data needs to be copied and what is the minimum number of data movement operations required, and finally do the data movement operations. Since this can all be done asynchronously, this set of operations can potentially be all done before the currently running task is over and the next task can start as soon as the current one is done. By cooperating with the runtime scheduler, the memory system can adapt to the system state at runtime and hide data movement latency effectively independently of current workload distribution.

- **Dynamic memory allocator** – The heterogenous dynamic memory manager should itself follow the same 4 characteristics of modern parallel memory allocators: speed, scalability, false share avoidance and low fragmentation. The proposed memory allocator organizes the memory hierarchically into blocks, super-blocks and hyper-blocks. Each processor core has its private heap and share the same shared pool. If a core does not have free blocks in its private heap, fetches new blocks (grouped into a super-block) from the shared pool. This private heap model paired with the fetching of new blocks from the shared pool in groups of super-blocks guarantees scalability by reducing synchronization points. When a deallocation request is received, if a heap is able to have a free super-block, that super-block is then returned to the shared pool. This mechanism aims to reduce fragmentation by recycling free space as soon as possible. Hyper-blocks, resulting from allocating sizes bigger than a super-block (i.e. custom size super-blocks), despite being stored in a specific structure are also returned to the shared heap as soon as they are available for recycling. A "lock less" oriented im-

plementation is essential to achieve speed and having blocks, super-blocks and hyper-blocks sizes as multiples of the DSM memory pages guarantees no false sharing.

When tested and benchmarked the proposed heterogeneous memory system showed mixed results. The parallel memory allocator using only CPU threads showed great results when compared to the GNU memory allocator. The allocator not only showed great scalability results but was significantly faster than the GNU memory allocator. Despite the CPU only results, when adding GPUs to the mix the average allocation time increased tremendously. The GPU threads allocation time was significantly higher than CPU threads. Since the DSM follows a central memory model, all allocation requests had to check for free memory and reserve it in host memory. These requests must go through PCI-Express and the latency introduced by the bus proved to be too costly. This, however, does not invalidates the proposed model (having multiple independent heaps for each processing core), but the central memory implementation was a deal breaker. A distributed memory model would allow a significant decrease in allocation times by removing the need to check for available space on remote memory. Unfortunately, given the simple nature of modern accelerators, the centralized model introduces the least overhead and stands as the best solution for the proposed heterogenous DSM.

Since the *GAMA* framework itself uses the memory allocator to manage dynamic structures (runtime task creation, work queues, etc.), one obvious concern is how is the framework's performance being affected by the memory allocator. Fortunately, all the framework's managing and bookkeeping operations are executed on the CPU side. Since the memory allocator's performance issues only arouse when used by GPU threads, the framework itself is not negatively affected. Actually, given that *GAMA* spawns several threads to manage the system, the framework is better off with the proposed memory allocator than with the GNU memory allocator.

The DSM results were significantly more encouraging. The DSM was benchmarked against three different algorithms, SAXPY, all-pairs N-Body simulation and Barnes-Hut N-Body simulation algorithms. SAXPY achieved the worst results of the three algorithms when using the DSM. The algorithm has several char-

acteristics that made it a terrible candidate for the DSM – no data reutilization, low ratio between computation and memory accesses and required more arithmetic operations to translate memory addresses than to do the actual SAXPY operation. Despite the algorithm being an "worst case scenario" for the DSM, when suiting the system with more than one GPU, the DSM showed improved results and in a particular problem size the DSM version was faster than the no DSM one. Being a memory bound algorithm, the contention introduced on PCI-E bus by the GPUs is a scalability bottleneck. The DSM however proved to be more efficient managing data across the PCI-E bus than the Nvidia's driver.

The all-pairs N-Body algorithm is more "cache friendly" since it has a regular memory access pattern and has data reutilization. Being a good algorithm for both the hardware cache (of the GPUs) and software cache (of the DSM), this algorithm stood as good benchmark of both caches. The performance of the DSM version was worse than the no DSM version for small input sizes, but was significantly faster for large input sizes. Unfortunately, the overhead introduced by the DSM is still significant when compared to an hardware cache and for small input sizes, with a "cache friendly" algorithm, the hardware cache is big and efficient enough to not justify the use of a software cache. For large input sizes, however, the additional space provided by the cache of the DSM is significant and the DSM is able to better mask data movement latency and address translation overhead with effective computation.

If the SAXPY algorithm is the DSM "worst case scenario" the Barnes-Hut algorithm is definitely the best. The DSM version was significantly faster than the no DSM version for the multiple problem sizes and hardware configurations. The reason is the irregular nature of the algorithm. This irregularity translates into poor hardware memory coalescing and high cache miss ratio. By moving the data to faster memory (device memory vs host memory) the cache miss penalty is significantly less time consuming and by having the prefetching and coalescing mechanism tuned for the specific algorithm (due to the users data methods) the DSM can use its software cache much more efficiently than the hardware cache. The DSM version superiority is more obvious when adding multiple GPUs. While the no DSM version problems are aggravated by having multiple GPUs, the DSM version achieved almost linear scalability results on average (linear in some cases).

Overall the DSM performance is positive. It had good results for the all-pairs algorithm despite requiring a large enough problem size to mask the overhead of the DSM (memory copies latency and address translation overhead). The Barnes-Hut algorithm was an all out winner and showed the advantage of using a system like this for irregular algorithms. The impact in performance of not using the accelerator memory on irregular algorithms can be significant and can be a scalability bottleneck. The DSM solves both problems and with no additional coding effort from the programmer. SAXPY achieved the worst results of all algorithms, but given the nature of the algorithm, that was to be expected. However, even with an algorithm of this nature, the DSM proved to be useful to improve scalability.

6.2 Future work

The obvious candidate for rework is the memory allocator. Despite the bad allocation times with GPUs, the proposed model produced great results with CPUs only. The centralized memory model encouraged by the DSM was not suitable for the memory allocator. Unfortunately, a redesign of the memory allocator to allocate memory at the speed of device's private memory on main memory, requires a shift to a more distributed model of DSM as well. Be as it may, any solution of this kind would require significant administrative code running on the accelerator managing distributed tables and incoming and outgoing messages queues. Not only this is high cyclomatic complexity code but, allied with the lack of fine-grain thread management, might proved to be a challenge to implement efficiently. GPU architectures seem to be increasingly more complete in terms of general purpose features in each generation and features required for a potential distributed memory manager might not be very far away. For example, new AMD GCN processors support page fault signals that could be used to detect software cache misses and trigger a thread to fetch the required page. Unfortunately, currently, that feature is not available to programmers but could be on the near future.

Another aspect that requires improvement is the job's two data dependencies methods for read-only and read/write data. Although important and in some

cases (like irregular algorithms) might prove crucial to achieve performance, at this point it stands as a requirement. One possible future feature of the DSM would be the ability to generate these methods automatically. This capability requires a compiler capable of doing some sort of pointer analysis to statically evaluate the data dependencies. The main goal would be to decrease the learning curve of new programmers to the framework and make it more flexible. By making the coding of a data dependencies method optional, the user coding process could be further expedited and leaves the requirement of a custom method only if the automatic generated one proved to be inefficient.

References

[1] Ahuja, S., Carriero, N., and Gelernter, D. Linda and friends. *Computer 19*, 8 (1986), 26–34.

[2] Augonnet, C., Clet-Ortega, J., Thibault, S., and Namyst, R. Data-aware task scheduling on multi-accelerator based platforms. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on* (2010), IEEE, pp. 291–298.

[3] Augonnet, C., and Namyst, R. A unified runtime system for heterogeneous multi-core architectures. In *Euro-Par 2008 Workshops-Parallel Processing* (2009), Springer, pp. 174–183.

[4] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience 23*, 2 (2011), 187–198.

[5] Barnes, J., and Hut, P. A hierarchical O(N log N) force-calculation algorithm. *nature 324* (1986), 4.

[6] Bennett, J., Carter, J., and Zwaenepoel, W. *Munin: Distributed shared memory based on type-specific memory coherence*, vol. 25. ACM, 1990.

[7] Berger, E., McKinley, K., Blumofe, R., and Wilson, P. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices 35*, 11 (2000), 117–128.

[8] Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., and Zhou, Y. *Cilk: An efficient multithreaded runtime system*, vol. 30. ACM, 1995.

[9] Boyd, C. DirectCompute: Capturing the Teraflop. Microsoft.

[10] Burtscher, M., and Pingali, K. An efficient CUDA implementation of the tree-based Barnes-Hut N-Body algorithm. *GPU Computing Gems Emerald Edition* (2011), 75.

[11] Chapman, B., Jost, G., and Van Der Pas, R. *Using OpenMP: Portable shared memory parallel programming*, vol. 10. MIT press, 2007.

[12] Coulouris, G. F., Dollimore, J., and T., K. *Distributed systems: Concepts and design*, 1 ed. Addison-Wesley Publishing Company, 1993.

[13] Diamos, G., and Yalamanchili, S. Harmony: An execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th international symposium on High performance distributed computing* (2008), ACM, pp. 197–200.

[14] Eisenberg, M. A., and McGuire, M. R. Further comments on Dijkstra's concurrent programming control problem. *Commun. ACM 15*, 11 (Nov. 1972), 999–.

[15] Fleisch, B., and Popek, G. *Mirage: A coherent distributed shared memory design*, vol. 23. ACM, 1989.

[16] Gelado, I., Stone, J., Cabezas, J., Patel, S., Navarro, N., and Hwu, W. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *ACM SIGARCH Computer Architecture News* (2010), vol. 38, ACM, pp. 347–358.

[17] Huang, X., Rodrigues, C., Jones, S., Buck, I., and Hwu, W. Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on* (2010), IEEE, pp. 1134–1139.

[18] Keltcher, C., McGrath, K., Ahmed, A., and Conway, P. The AMD Opteron processor for multiprocessor servers. *Micro, IEEE 23*, 2 (2003), 66–76.

[19] Kurd, N., Douglas, J., Mosalikanti, P., and Kumar, R. Next generation Intel micro-architecture (Nehalem) clocking architecture. In *VLSI Circuits, 2008 IEEE Symposium on* (2008), IEEE, pp. 62–63.

[20] Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. The Stanford dash multiprocessor. *Computer 25*, 3 (1992), 63–79.

[21] Li, K., and Hudak, P. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS) 7*, 4 (1989), 321–359.

[22] Linderman, M., Collins, J., Wang, H., and Meng, T. Merge: A programming model for heterogeneous multi-core systems. In *ACM SIGOPS Operating Systems Review* (2008), vol. 42, ACM, pp. 287–296.

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

[24] Mantor, M., and Houston, M. AMD Graphic Core Next: Low power high performance graphics & parallel compute. In *presentation, High-Performance Graphics Conf* (2011).

[25] Mark, W. R., Glanville, R. S., Akeley, K., and Kilgard, M. J. Cg: A system for programming graphics hardware in a C-like language. *ACM Trans. Graph. 22*, 3 (July 2003), 896–907.

[26] Michael, M. Scalable lock-free dynamic memory allocation. In *ACM Sigplan Notices* (2004), vol. 39, ACM, pp. 35–46.

[27] Moore, G., et al. Cramming more components onto integrated circuits. *Proceedings of the IEEE 86*, 1 (1998), 82–85.

[28] Nickolls, J., and Dally, W. The GPU computing era. *Micro, IEEE 30*, 2 (2010), 56–69.

[29] Nitzberg, B., and Lo, V. Distributed shared memory: A survey of issues and algorithms. *Computer 24*, 8 (1991), 52–60.

[30] Nvidia, C. NVIDIA CUDA programming guide 4.2, 2012.

[31] Nyland, L., Harris, M., and Prins, J. Fast N-Body simulation with CUDA. *GPU gems 3* (2007), 677–695.

[32] Pheatt, C. Intel threading building blocks. *Journal of Computing Sciences in Colleges 23*, 4 (2008), 298–298.

[33] Protic, J., Tomasevic, M., and Milutinovic, V. A survey of distributed shared memory systems. In *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on* (1995), vol. 1, IEEE, pp. 74–84.

[34] Rost, R., Kessenich, J., and Lichtenbelt, B. *OpenGL shading language*. Addison-Wesley Professional, 2004.

[35] Shi, W., Hu, W., and Tang, Z. An interaction of coherence protocols and memory consistency models in dsm systems. *ACM SIGOPS Operating Systems Review 31*, 4 (1997), 41–54.

[36] Stenstrom, P. A survey of cache coherence schemes for multiprocessors. *Computer 23*, 6 (1990), 12–24.

[37] Stone, J., Gohara, D., and Shi, G. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering 12*, 3 (2010), 66.

[38] Sutter, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal 30*, 3 (2005), 202–210.

[39] Wang, P., Collins, J., Chinya, G., Jiang, H., Tian, X., Girkar, M., Yang, N., Lueh, G., and Wang, H. EXOCHI: Architecture and

programming environment for a heterogeneous multi-core multithreaded system. *ACM SIGPLAN Notices 42*, 6 (2007), 156–166.

[40] Wilkes, M. The memory gap and the future of high performance memories. *SIGARCH Comput. Archit. News 29* (March 2001), 2–7.

[41] Wittenbrink, C., Kilgariff, E., and Prabhu, A. Fermi GF100 GPU architecture. *Micro, IEEE 31*, 2 (2011), 50–59.

[42] Wolfram, G. Dynamic memory allocator implementations in linux system libraries.

[43] Zhou, S., Stumm, M., and McInerney, T. Extending distributed shared memory to heterogeneous environments. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on* (1990), IEEE, pp. 30–37.