ON THE PROGRAMMABILITY OF MULTI-GPU COMPUTING SYSTEMS

by
Javier Cabezas Rodríguez

Advisor: Prof. Nacho Navarro
Co−advisor: Prof. Wen−mei Hwu

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Architecture

Universitat Politècnica de Catalunya
2015

Barcelona, Spain

Abstract

Multi-GPU systems are widely used in High Performance Computing environments to accelerate scientific computations. This trend is expected to continue as integrated GPUs will be introduced to processors used in multi-socket servers and servers will pack a higher number of GPUs per node. GPUs are currently connected to the system through the PCI Express interconnect, which provides limited bandwidth (compared to the bandwidth of the memory in GPUs) and it often becomes a bottleneck for performance scalability. Fortunately, the utilization of integrated GPUs that share the physical memory with CPUs, or higher-bandwidth interconnects such as NVLink, will help to address this problem.

Current programming models present GPUs as isolated devices with their own memory, even if they share the host memory with the CPU. Programmers need to copy the required data to the GPU memory before it is used in a GPU computation, and copy the generated results to host memory before they are accessed by the CPU code. Moreover, complex data transfer schemes (e.g., prefetching or speculative updates) are required to efficiently exploit GPUs, but they are difficult to maintain, especially across different system topologies. This problem is aggravated when using multiple GPUs. Multi-GPU systems are programmed like distributed systems in which each GPU is a node with its own memory. Programmers explicitly manage allocations in all GPU memories and use primitives to communicate data between GPUs. Furthermore, programmers are required to use mechanisms such as command queues and inter-GPU synchronization. This explicit model harms the maintainability of the code and introduces new sources for potential errors.

The first proposal of this thesis is the HPE model. HPE enables multi-GPU applications to efficiently exchange data while preserving developer productivity and application maintainability. HPE builds a simple, consistent programming interface based on three major features. (1) All device address spaces are combined with the host address space to form a Unified Virtual Address Space, or UVAS. (2) Programs are provided with an Asym-

metric Distributed Shared Memory (ADSM) system for all the GPUs in the system. It allows applications to allocate memory objects that can be accessed by any GPU or CPU. HPE exploits the UVAS to easily keep track of the location and the state of the copies for each object. (3) Every CPU thread can request a data exchange between any two GPUs, through simple memory copy calls. Such a simple interface allows HPE to automatically optimize data exchanges between devices; eliminating the need for application code to handle different system topologies.

Experimental results show that the HPE model eases programming of multi-accelerator applications while providing performance improvements of 2× compared to the best data transfer scheme implemented on top of CUDA 3. Improvements on real applications range from 5% in compute-bound benchmarks and up to 2.6× in communication-bound benchmarks. HPE transparently implements sophisticated communication schemes that can deliver up to a 2.9× speedup in I/O device transfers.

The second proposal of this thesis is a shared memory programming model that exploits the new GPU capabilities for remote memory accesses to remove the need for explicit communication between GPUs. This model turns a multi-GPU system into a shared memory system with NUMA (i.e., Non-Uniform Memory Access) characteristics. In order to validate the viability of the model we also perform an exhaustive performance analysis of remote memory accesses over PCIe. We show that the unique characteristics of the GPU execution model and memory hierarchy help to hide the costs of remote memory accesses.

Results show that PCI Express 3.0 is able to hide the costs of a 10% of remote memory accesses if the kernels produce high GPU core occupancy and the accesses are spread through the whole kernel execution. We also show that caching of remote memory accesses can have a large performance impact on kernel performance.

Finally, we introduce AMGE, a programming interface, compiler support and runtime system that automatically executes computations that are programmed for a single GPU across all the GPUs in the system. The programming interface provides a data type for multidimensional arrays that allows for robust, transparent distribution of arrays across all GPU memories. The compiler extracts the dimensionality information from the type of each array, and is able to determine the access pattern in each dimension of the array. The runtime system uses the compiler-provided information to automatically choose the best computation and data distribution configuration to minimize inter-GPU communication and memory footprint. This model effectively frees programmers from the task of decomposing and distributing computation and data to exploit several GPUs.

AMGE achieves almost linear speedups for a wide range of dense computation benchmarks on a real 4-GPU system with an interconnect with moderate bandwidth. We show that irregular computations can also benefit from AMGE, too, if kernels do not suffer from computation distribution imbalance across thread blocks. We believe that AMGE can be used in shared memory multi-GPU systems to automatically scale the performance of GPU kernels to multiple GPUs.

Thanks to the results obtained in the experiments of this thesis we demonstrate that careful programming interface design and efficient implementation of our proposals enable applications with not only simpler and more maintainable codebases, but also a performance scalability similar to hand-tuned low-level implementations especially designed for multi-GPU systems.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In the last years, massively-parallel accelerators have experienced widespread adoption in different environments due to their outstanding performance in many types of computations. Companies and research centers that rely on High Performance Computing (HPC) to solve complex scientific simulations were the first to embrace the utilization of such processors. Further improvements in the manufacturing technologies of chips have also made possible the integration of accelerators and general purpose processors in a single (heterogeneous) chip; and they can now be found in mobile devices, desktops, and servers. Thus, this types of accelerators are becoming ubiquitous and this trend is expected to continue in the next years.

The most common type of massively-parallel accelerator is the Graphics Processing Unit (i.e., GPU). GPUs have traditionally excelled at executing fixed-function primitives that process in parallel data such as vertices, edges and pixels. With the introduction of user-defined shaders (user-defined computations on the geometry or color of 3D objects), GPUs were extended to be programmable processors. At the beginning, this capabilities were only accessible through extensions of the most popular graphics' application program interfaces (e.g., OpenGL and DirectX). Many scientists saw the potential of GPUs to be used for general purpose computing, and started to develop programming frameworks such as Lib Sh [67], BrookGPU [31], and Accelerator [90]. Since then, GPU vendors have created new programming languages and frameworks such as NVIDIA®CUDA™ [74] or OpenCL™ [56] to enable true general purpose computing on GPUs.

The preferred form factor in HPC environments is *discrete* GPUs because they provide their own specialized memories (e.g., GDDR5) that deliver much higher bandwidth than host memory. Examples of discrete GPUs are NVIDIA Tesla and AMD FirePro. Discrete GPUs are connected to the

system through standard I/O interconnects like PCI Express. Many manufacturers also couple a general purpose processor and an *integrated* GPU in the same die. This type of chip is common in desktop and mobile systems. Examples of integrated CPU/GPU processors include Intel Haswell, AMD Kaveri, and NVIDIA K1. In these systems, CPUs and GPUs share the host memory. This has two effects: (1) Integrated GPUs have lower performance than their discrete counterparts (due to the lower memory bandwidth). (2) On the other hand, they benefit from faster CPU↔GPU communication since they do not rely on I/O interconnects.

Oftentimes, problems are too computationally demanding or the required data is too big to be run on a single GPU. In such cases, the problem is decomposed into smaller subproblems (e.g., using domain decomposition) that are executed on several GPUs in parallel. Many HPC systems systems install several discrete GPUs per node[1]. GPUs installed in the same node benefit from lower-latency and higher-bandwidth communication. In order to pack as many GPUs as possible in the same node, some manufacturers have announced products that include several GPUs in the same board (e.g., NVIDIA Pascal). Nevertheless, the increased performance of multiple GPUs comes at the cost of higher programming complexity.

Current programming models present GPUs as isolated devices with their own memory. This is the case even if they share the host memory with the GPU, because most chips with integrated GPUs do not implement hardware memory coherence between the CPU and the GPU cores. However, exposing separate memories hurts programmability. Programmers need to copy the required data to the GPU memory before it is used in a GPU computation, and copy the generated results to host memory before they are accessed by the CPU code. Moreover, simple data transfer schemes are likely to incur into performance penalties. On the other hand, elaborated schemes (e.g., prefetching or speculative updates) require complex implementations that are difficult to maintain, especially across different system topologies.

This problem is aggravated when using multiple GPUs. Multi-GPU systems are programmed like distributed systems in which each GPU is a node with its own memory. Programmers need to manage allocations in all GPU memories and explicitly use primitives to communicate data between GPUs, which can be difficult in computations with complex data dependencies between GPUs. Furthermore, in order to manage several GPUs efficiently, programmers are required to use mechanisms such as command queues and inter-GPU synchronization. This, again, further harms the maintainability

[1]No integrated multi-GPU systems have been released so far, but they are expected to appear in the next years.

of the code and introduces new sources for potential errors.

1.1 Objectives and contributions

The main objective of this thesis is to facilitate the utilization of GPUs so that they can be adopted by a broader community. More specifically, improvements to the current programming models for GPUs are proposed in order to simplify the management of multiple GPUs and the data transfers between the host and GPU memories. Some of these changes require support from the Operating System or the GPU architecture, too.

1.1.1 Multi-GPU Reverse Time Migration

In order to accomplish our goals, we first analyze the GPU programmability issues in real applications. The RTM (i.e., Reverse Time Migration) application is used as a guiding example in this thesis. It is a key application in the oil and gas industry with much higher computing power requirements than previous algorithms, that can only be fulfilled thanks to multi-GPU execution.

We present an optimized implementation of the RTM that is specifically designed to exploit the architectural characteristics of the GPUs. The work is divided in two parts:

1. An optimized implementation of the different computational kernels on the GPU.

2. A multi-GPU multi-node execution framework for RTM. This work was done using CUDA 3, which imposes several restrictions on how CPU threads can access the GPUs in the system (e.g., each CPU thread can access 1 GPU only).

This implementation is measured against three reference HPC platforms: one that employs traditional cache-coherent cores, another one that uses Cell B.E. accelerators [53], and a Convey HC-1 FPGA system. Results show that GPUs outperform any other accelerator. However, PCIe and disk I/O needs to be carefully performed to overlap them with computation to hide the costs. Limitations in the GPU programming model makes this a difficult task, and we identify several programmability issues that we try to address in the thesis.

1.1.2 Data sharing between the CPU and multiple GPUs

To effectively utilize a multi-GPU computing system, application developers need to distribute large data sets across GPU memories. Within a heterogeneous computing cluster, application-level data exchange between GPUs involves interactions between node-level APIs such as CUDA or OpenCL, and inter-node APIs such as MPI. While MPI hides the complexity of the system interconnect topology and inter-node routing, there is currently no similar node-level interface. Therefore, developers are left with the challenging task of code versioning needed to effectively use a wide variety of hardware and drivers with very different capabilities.

Moreover, CPU and GPU have separate memory spaces and developers are in charge of managing separate copies on host and GPU memories for each data structure, and keeping them coherent through explicit memory transfers. Some solutions propose a single virtual memory space that is shared between CPU and GPU [45, 52]. In these solutions, each object is allocated once although memory is allocated both in host and GPU memory; and a runtime system transparently keeps the different copies of the object coherent. However, they target systems with a single GPU.

We propose the Heterogeneous Parallel Execution (HPE) programming interface and its runtime system. HPE enables multi-GPU applications to efficiently exchange data while preserving developer productivity and application maintainability. HPE is available to CUDA and OpenCL developers as a user-level library. It allows applications to allocate memory objects that can be accessed by any GPU or CPU in the system. The HPE runtime transparently handles data copies.

Both CUDA and OpenCL allow applications to interact with all devices in the system. However, the physical bus topology of the system dictates the level of interaction and data exchanges allowed between GPUs. As a result, programmers write application code to discover the physical system topology and implement different code paths for each type of system configuration. This represents a significant effort resulting in large, hard-to-maintain application code base.

HPE builds a simple, consistent programming interface based on three major features.

1. All GPU address spaces are combined with the host address space to form a Unified Virtual Address Space, or UVAS. Starting with the Fermi generation, all NVIDIA GPUs support UVAS based on virtual memory.

2. Programs are provided with an Asymmetric Distributed Shared Mem-

ory (ADSM) system [45] for all the GPUs in the system. HPE exploits the UVAS to easily keep track of the location and the state of the copies for each object. The simple relation between a GPU address and its corresponding host address allows a lock-free implementation of the HPE runtime.

3. Every CPU thread can request a data exchange between any two GPUs, through simple memory copy calls. The HPE runtime manages thread-to-GPU connection and performs intermediate data copies when necessary to realize this abstraction. Such a simple interface allows HPE to automatically optimize data exchanges between GPUs; eliminating the need for application code to handle different system topologies.

We have been working with NVIDIA to include many of the features of HPE in the latest versions of CUDA.

1.1.3 Shared memory programming for multiple GPUs

The common approach to multi-GPU programming is to use each GPU as an isolated device with its own memory. This imposes the utilization of explicit memory transfers between GPU memories every time a GPU needs data that is located on the memory of a different GPU. Further, these memory transfers must be overlapped with computation to minimize their overhead, thus increasing the complexity of the code. While HPE takes care of some of these optimizations, we would like to remove all explicit data transfers.

Modern NVIDIA GPUs provide the capability of accessing the memories of all the GPUs connected to the same PCI Express root complex [5]. They also present a single Unified Virtual Address Space (i.e., UVAS) that includes all the GPU memories in the system and the host memory. Thanks to these features, CUDA kernels can access any memory in the system through regular `load`/`store` instructions. While these features have been available for some time they have gone unnoticed among application developers, and their utilization has been mainly restricted to accelerate bulk data transfers between GPU memories and to enable better integration with I/O devices.

In this dissertation, we propose GPU-SM, a shared memory programming model that exploits the new GPU capabilities to remove the need for explicit communication between GPUs. The memory of multiple GPUs is aggregated to form a shared memory system with NUMA (i.e., Non-Uniform Memory Access) characteristics. In this type of systems computation can be freely decomposed and distributed across all the GPUs in the system, because any GPU can access all the memory in the system, although with a lower bandwidth and higher latency.

We also perform an exhaustive performance analysis of remote memory accesses over PCIe and their viability as a mechanism to implement the shared memory model in multi-GPU systems. Different PCIe revisions (i.e., 2.0 and 3.0) and topologies are studied. We show that the highly-multi-threaded GPU execution model helps to hide the costs of remote memory accesses. Other features introduced in the latest GPU families such as read-only caches can minimize the amount of accesses to remote GPUs. The importance of the thread block scheduling and its relation with data decomposition is also discussed, as they determine how remote memory accesses are distributed along kernel execution.

1.1.4 Transparent parallelization of GPU programs to run on multiple GPUs

While using a shared memory model improves programmability of multi-GPU systems, programmers are still in charge of decomposing the problem and distributing it across several GPUs. Moreover, memory placement needs to be performed in such a way that remote memory accesses are minimized, in order to avoid the impact on the performance. This has shown to be a challenging task in the past in other type of systems [30, 66, 65, 38] and usually rely on hardware mechanisms not available in current GPUs such as paging..

In this dissertation, we propose exposing to programmers a single virtual GPU that aggregates the resources of all GPUs in the system. Similar solutions have been proposed [57, 61], but their design imposes fundamental limitations. (1) Memory footprint overhead: these solutions replicate portions of the arrays that are never accessed. This is because they do not take into account the dimensionality of the arrays. For example, consider a kernel that performs n-dimensional tiling (a common pattern in dense GPU computations [83, 92, 21]) where each computation partition accesses a non-contiguous memory region of a matrix. In such a case, they transfer the whole memory address ranges accessed by each computation partition, which may include large portions of the array that are never used. This limits the size of the problems that can be handled and imposes performance overheads due to larger data transfers. (2) Data coherence overhead: replicated output memory regions need to be merged in the host memory after each kernel call. In many cases, this merge step leads to large performance overheads. (3) Applications that use atomic and global memory instructions, resort to single-GPU execution.

In this thesis, we introduce AMGE (Automatic Multi-GPU Execution), a

programming interface, compiler support and runtime system that automatically executes computations that are programmed for a single GPU across all the GPUs in the system. The programming interface provides a data type for multidimensional arrays that allows for robust, transparent distribution of arrays across all GPU memories. The compiler extracts the dimensionality information from the type of each array, and is able to determine the access pattern in each dimension of the array. The runtime system uses the compiler-provided information to automatically choose the best computation and data distribution configuration to minimize inter-GPU communication and memory footprint.

AMGE assumes non-coherent non-uniform shared memory accesses (NCC-NUMA) between GPUs through a relatively low-bandwidth interconnect, such that all GPUs can access and cache any partition of the arrays. Thus, we ensure that arrays can be arbitrarily decomposed, distributed and safely accessed from any GPU in the system. In current systems based on discrete GPUs, we utilize Peer-to-Peer [5] and Unified Virtual Address Space [74] technologies that enable a GPU to transparently access the memory of any other GPU connected to the same PCIe domain. While remote GPU memory accesses have been used in the past [89], this is the first work to use them as an enabling mechanism for automatic multi-GPU execution.

Results show that AMGE can effectively provide linear speedups for a wide reange of dense computations a 4 GPU system. Moreover, the size of the problems that can be handled is much larger than previous solutions. Furthermore, AMGE can also be beneficial to irregular workloads.

1.2 Organization

This dissertation is organized as follows:

Chapter 1: Introduction Provides an overview of the programmability problems we are trying to solve and outline the proposed solutions.

Chapter 2: Reference Hardware and Software Environment The chapter explains the unique characteristics of the GPU architecture and the technologies (e.g., PCI Express, GDDR5 memory) in multi-GPU systems, that are used in the proposals. This is followed with a discussion of current programming models for GPUs, their features, and their limitations. Since the RTM algorithm is used as a guiding example, it is introduced here, too.

Chapter 3: State of the Art Several works have been proposed to simplify the programmability of GPU-based systems. These previous works are discussed and their limitations are highlighted. We focus on memory management and automatic multi-GPU execution.

Chapter 4: Guiding Example: Reverse Time Migration Reverse Time Migration (RTM) is used as a guiding example in this thesis to illustrate programmability problems in multi-GPU systems. The RTM computation is composed of a number of computational kernels and has high I/O requirements. We explain the implementation optimizations for GPUs and compare them with previous approaches for different architectures.

Chapter 5: Heterogeneous Multi-GPU Execution Multi-GPU execution is required to handle real-world problem sizes. This chapter introduces a multi-GPU paralelization strategy in CUDA for RTM with support for several nodes through MPI. We identify and analyze the programmability problems that arise and propose the HPE model that eliminates or minimizes them.

Chapter 6: Shared Memory GPU Programming Discusses the hardware requirements and the characteristics to enable shared memory programming in multi-GPU systems. The chapter also includes an analysis of the performance characteristics of PCI Express and the capabilities of current GPUs to hide the costs of remote memory accesses.

Chapter 7: Automatic Multi-GPU Execution This chapter introduces a programming framework that transparently decomposes computation and data and distributes them across all the GPUs in the system.

Chapter 8: Conclusions and Future Work Sumarizes the discoveries made in this thesis and outlines future directions to achieve simpler and more efficient multi-GPU execution.

Chapter 2

Reference Hardware and Software Environment

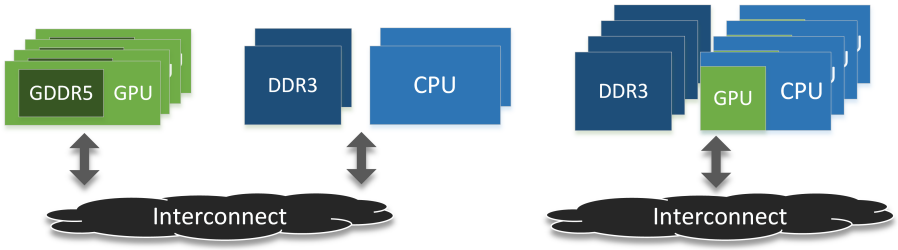
This thesis proposes improvements in the programmability of systems that install several GPUs. This chapter introduces the details of the GPU hardware and programming models which are important to understand the performance trade-offs and the implementability of our proposals. It also presents the terminology used in the rest of the thesis.

2.1 Graphics Processing Units

Modern GPUs are presented in two different form factors.

- *Discrete* GPUs are dedicated boards connected to the rest of the system through expansion slots such as PCI Express (shown in Figure 2.1a). They have their own high-bandwidth memory, which is explicitly managed by the programmers.

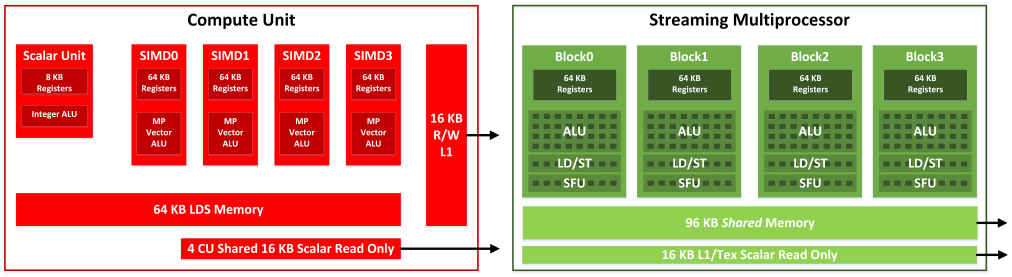
- *Integrated* GPUs are included in the same die as the CPU and both share the host memory (shown in Figure 2.1b). They may also share some levels of the cache memory hierarchy with the CPU cores. For example, the CPU and GPU in the Intel Ivy Bridge [49] chip share the L3 cache. AMD APU [2] provides both coherent (which shares the L2 cache) and non-coherent interconnects for the integrated GPU, which are chosen at programmer's request. On the other hand, the GPU in the NVIDIA K1 [14] chip does not share any level of the cache memory hierarchy with the CPU cores.



(a) Discrete (b) Integrated

Figure 2.1: Multi-GPU system architectures.

Discrete designs are preferred in High Performance Computing environments due to their much higher memory bandwidth. Moreover, several discrete GPUs can be installed in each node, while there are no multi-GPU designs that use integrated GPUs yet. Since this thesis targets the programmability of multiple GPUs, we mainly focus on discrete GPUs, although many of the presented concepts and analyses can be applied to integrated GPUs, too.



(a) AMD Graphics Core Next (b) NVIDIA Maxwell

Figure 2.2: Microarchitecture of AMD and NVIDIA GPU cores.

Current GPUs have several highly-multithreaded cores that contain wide vector units. They use a SIMT (i.e., Single Instruction Multiple Threads) execution model in which threads are organized in fixed-length groups (i.e., warps in NVIDIA GPUs, wave fronts in AMD GPUs) that execute the same instruction concurrently on the different lanes of the vector units. The current size of a warp is 32 threads in NVIDIA GPUs and 64 threads in AMD GPUs. When threads in the same warp take different execution paths they produce thread divergence. Since all threads in the warp execute in lockstep, each path is executed by all the threads, although predication is used to disable the lanes of the threads that are not in the executed path. Divergence degrades the throughput of the GPU up to as many times as the number

of diverging execution paths and must be minimized. GPU cores switch execution between different warps to hide the cost of long latency operations such as memory requests. NVIDIA refers to each of this cores as *Streaming Multiprocessor* (i.e., SM) and uses *CUDA core* to refer to each of the lanes of the vector unit [46, 4, 12] (Figure 2.2b). AMD uses *Computing Unit* (i.e., CU) for the cores, *SIMD* for the vector unit and *work item* for the vector lane [1] (Figure 2.2a). In this thesis we use the terms warp, GPU core, vector unit and vector lane.

GPUs typically include a cache memory hierarchy, but it serves a different purpose than CPU caches. In CPUs, threads compute coarse grain tasks and CPU caches exploit spatial/temporal locality within each thread to minimize the access latency by reducing the number of off-chip memory requests. GPUs, however, use a large number of threads that perform fine grain computations which result in much lower data locality within each thread. On the other hand, neighboring threads tend to access to contiguous data elements. Thus, the main purpose of caches in GPUs is to serve as a coalescing point: memory accesses from different threads in a warp to elements within the same cache line are *coalesced* into a single request. In order to maximize performance, data must be laid out in such a way that memory accesses are always coalesced. A scratchpad memory is also commonly provided for programmers to exploit inter-thread locality.

NVIDIA GPUs implement a two-level hierarchy with a first level composed of a non-coherent private cache per core, and a shared second level cache. L1 caches are write-through and, therefore, modifications from different cores to the same cache line are consolidated in the L2 cache. The cache hierarchy supports atomic operations, which are implemented in the L2 cache, too. The Kepler family of GPUs added a second private cache per SM for read-only data. Each core also contains a scratchpad memory (96 KB in the Maxwell family, 48 KB in previous families). AMD Core Next architecture use a similar two-level memory hierarchy with a local 64 KB scratchpad per core. The main difference with respect to NVIDIA GPUs is that L1 caches are coherent. AMD GPUs also provide a scalar core that handles the execution of common scalar operations (e.g., branches, pointer arithmetic) across threads in a wave front and has its own L1 read-only cache.

2.1.1 Memory technologies

CPUs use DDR SDRAM (i.e., Double Data Rate Synchronous Dynamic Random Access Memory) or simply DDR. The revision of this technology that is currently used is DDR3, although many vendors are already shipping DDR4 modules. High-end systems provide several independent communi-

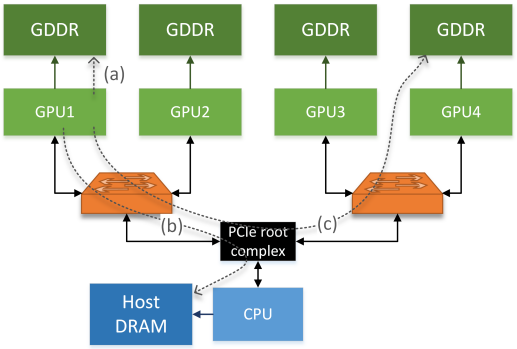


Figure 2.3: Multi-GPU NUMA system targeted in this thesis.

cation channels with the memory modules (AMD Socket G34 supports up to 4 channels). Each channel uses a 64-bit bus and many channels can be combined to build a wider bus (i.e., *ganged* mode) although this mode is not commonly used. Read and write accesses to the SDRAM are burst oriented; an access starts at a selected location and consists of a total of four or eight data words. For write operations a mask can be used to specify which individual bytes are actually written to memory.

Conversely, discrete GPUs use GDDR5 (i.e., Graphics DDR version 5), which presents some differences with respect to DDR3. GDDR5 devices are always directly soldered down on the board and are not mounted on a memory module (e.g., DIMM). They operate at a much higher frequency than DDR devices. Each channel uses a 32-bit bus, and channels are always ganged by the GPU memory controller to form a much wider bus (up to 512 bits). Bursts in GDDR5 always have a size of eight data words.

2.2 Multi-GPU systems

The base system targeted in this thesis is composed of several discrete GPUs connected to the system through a PCI Express (i.e., PCIe) interconnect (Figure 2.3). Besides having access to their own memories, since the Fermi microarchitecture [46], NVIDIA GPUs can access other memories through the PCIe interconnect (i.e., GPUDirect[5]) without host code intervention. At the same time, they introduced a single Unified Virtual Memory Address Space (i.e., UVAS) for all the GPUs in the system. The goal of UVAS is to allow every object in the system, no matter which physical memory it resides in, to have a unique virtual address to be used by application pointers. Combining these two features allows regular `load`/`store` instructions to transparently generate local or remote requests, based on the translation of

virtual address to the physical location of the data being accessed. Several of the designs proposed in this thesis take advantage of the remote memory access mechanism to implement shared memory programming on multiple GPUs. As far as we know, AMD does not currently support remote memory access between GPU memories.

2.2.1 Interconnect

Practically all current discrete GPUs use the PCIe interconnect to interface with the host. However, interconnects that offer higher bandwidth have been announced.

PCI Express

PCIe is a expansion bus that uses a point-to-topology that connects the devices to the *root complex*[76]. It uses a separate link per device, which can contain several lanes (currently from 1 to 16 lanes are commonly used). The root complex can interface with PCIe endpoints (i.e., devices) or switches that multiplex the link among several devices. This flexibility allows to create complex device hierarchies. Data is transmitted in packets, that can be stripped accross lanes. Therefore, bandwidth increases with the number of lanes. Moreover, PCIe supports full-duplex communication between any two endpoints. Current PCIe 3.0 provides up to 16 GB/s per direction. The encoding used to transmit data in previous PCIe versions reduced the effective bandwidth by a 20%, but PCIe 3.0 uses a different encoding that introduces a 2% overhead, only.

The PCIe standard packets support up to 30-byte headers and up to 4 KB payloads (although most PCIe controllers in current processors limit it to 256 bytes or less [50]). The overhead imposed by the size of the headers makes PCIe not well suited for very small transfers. On the other hand, transfers larger than the payload size are broken by the PCIe controller to several packets (which is more efficient than breaking the transfer to smaller transfers in software). Another source of overhead is due to PCIe transactions not being able to use pageable host memory. To solve this problem, the GPU driver copies data to temporary non-pageable (or *pinned*) buffers before transmitting them to the GPU memory. While double-buffering can be used to hide this cost, they are only effective for large data transfers. Figure 2.4 shows the achieved memory transfer rates for different data sizes in a system with a NVIDIA C2050 and a PCIe 2.0 x16 interconnect. Lines with circles represent transfers using pinned memory buffers instead of regular pageable allocations.

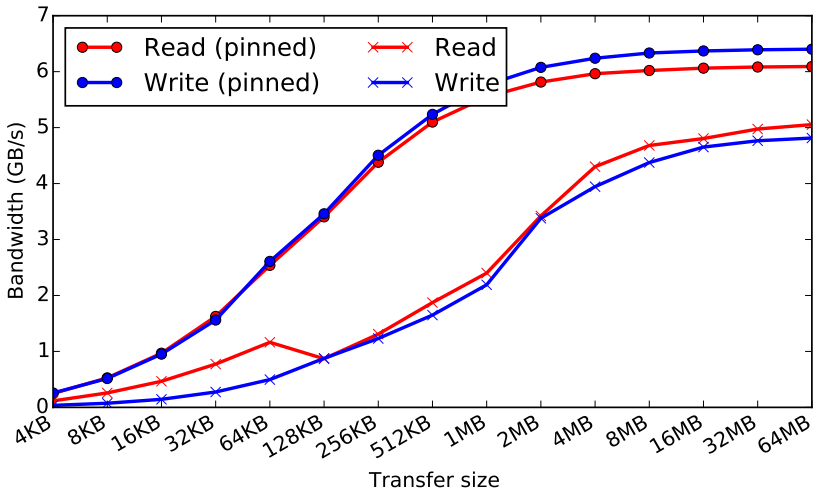


Figure 2.4: PCIe transfer rates for different data sizes.

Memory	**Latency**	**Bandwidth**
DDR3 (CPU, 2 channels)	$\sim$ 50 ns	$\sim$ 30 GBps
GDDR5 (GPU, 4-8 channels)	$\sim$ 500 ns	$\sim$ 250 GBps
HBM	$\sim$ 500 ns	500–1000 GBps
Interconnect	**Latency**	**Bandwidth**
Hypertransport 3	$\sim$ 40 ns	25+25 GBps
QPI	$\sim$ 40 ns	25+25 GBps
PCIe 2.0	$\sim$ 200 ns	8+8 GBps
PCIe 3.0	$\sim$ 200 ns	16+16 GBps
NVLink (4 interconnection points)	-	80 GBps

Table 2.1: Memory and interconnection network characteristics.

NVLink

NVIDIA will introduce a new interconnect called NVLink [13] in the Pascal GPU family (to be released in 2016), in an effort to replace PCIe with a faster bus. NVLink also uses a point-to-point bus and includes one or several NVLink interconnection points in each device. Devices can bond together several interconnection points to increase the bandwidth. Preliminary studies show 80 GB/s per direction for a device using four interconnection points.

2.2.2 GPU NUMA systems

The interconnects and memories in the system have different different latency and bandwidth characteristics (Table 2.1). GPUs access their local memory (Figure 2.3, arc a) with full-bandwidth (e.g., $\sim$ 250 GBps in GDDR5). They

access other memories in the system through the PCIe interconnect: host memory (arc b) or another GPU memory (arc c)[1]. Remote accesses can traverse any PCIe switch found between the client and the server GPUs. Both CPU memory and the inter-GPU interconnects like PCIe deliver a memory bandwidth which is an order of magnitude lower than the local GPU memory (e.g., $\sim$ 12 GBps in PCIe 3), thus creating a Non-Uniform Memory Access (i.e., NUMA) system. Future interconnects such as NVLink will increase the bandwidth, but memory interfaces with higher bandwidths have been announced too (HBM [6] will deliver up to 1 TB/s).

Remote access characteristics

Besides the lower bandwidth and increased latency, remote accesses present some other peculiarities compared to regular memory accesses. They are not cached in the regular memory hierarchy because modifications from different GPUs to the same cache line could produce coherence problems. However, remote accesses to data that is only read in the computation can be safely cached in the read-only (R/O) cache (currently, this must be specified by programmers by using the `__ldg` intrinsic provided by CUDA).

2.3 Programming models for GPU-based systems

GPUs are typically programmed using a Single Program Multiple Data (SPMD) programming model, such as NVIDIA CUDA [74] or OpenCL [56]. Unless stated otherwise, the CUDA naming conventions are used in the rest of the thesis.

2.3.1 C for CUDA

Although the programming framework is called C for CUDA, we refer to it simply as CUDA in the rest of the thesis. CUDA [74] is divided into two parts:

1. A C/C++ API that exposes GPUs to programs and allows them to manage their memory and launch computations to be run on the GPUs.

[1] In systems with multiple CPU sockets, the inter-CPU interconnect (e.g., HyperTransport/QPI) might need to be traversed, too. Unfortunately, current systems do not support routing remote GPU↔GPU requests over the inter-CPU interconnect. Therefore, all the experiments presented in this thesis that require the remote memory access capability are performed on systems with a single CPU socket.

2. A C++-like programming language to write computations that run on the GPU.

CPU-GPU interface

Programmers need to manage GPU and host memories. Although they are presented with a UVAS, they control in which GPU memory is physically allocated. They need to copy the necessary data to the GPU memory before launching any GPU computation that uses it. Conversely, output data computed on a GPU needs to be copied back to host memory before it can be accessed by the CPU. Thanks to the UVAS, pointers to data allocated on one GPU can be passed to a different GPU, since it can be transparently accessed through the PCIe interconnect (although with much higher latency and much lower bandwidth).

The GPU is a passive device that executes asynchronous commands pushed by the host code: mainly kernel launches and memory transfers. The GPU address space is abstracted with a *CUDA context*. By default, a single CUDA context is assigned to each GPU, but additional contexts can be created. GPUs provide several command queues which are abstracted as *CUDA streams*. Commands pushed to a stream are executed in order, but commands from different streams can execute in any order and, if there are enough available resources, concurrently. Therefore, several streams must be used to overlap computation and data transfers. CUDA provides different levels of synchronization: device, stream, and individual operation (using *CUDA events*). Event barrier operations can be also pushed to streams to guarantee inter-stream command ordering. All these abstractions need to be used in most cases in order to achieve high performance, thus greatly increasing coding complexity.

GPU programming model

Programmers encapsulate computations in functions (called *kernels*) that are executed in parallel by a potentially large number of threads, although each thread might take a completely different control flow path within the kernel. All these threads are organized into a *computation grid* of groups of threads (i.e., thread blocks). Each thread block has an identifier and each thread has an identifier within the thread block, that can be used by programmers to map the computation to the data structures. CUDA provides a weak consistency model: memory updates performed by a thread block might not be perceived by other thread blocks, except for atomic and memory fence (GPU-wide and system-wide) instructions.

Each thread block is scheduled to run on an SM, and threads within a thread block are issued in fixed-length groups (the previously described warps). Each thread has its own set of private registers and threads within the same thread block can communicate through a shared user-managed scratchpad and using synchronization instructions. The number of thread blocks that can execute concurrently on the same SM (i.e., occupancy) depends on the number of threads and other resources needed by each block (i.e., scratchpad memory and registers). Hence, the utilization of these resources must be carefully managed to achieve full utilization of the GPU.

2.3.2 OpenCL

OpenCL [56] is an open programming framework maintained by the Khronos standardization organization. It shares many design principles with CUDA but its scope is far more ambitious since it targets virtually any type of computing device. However, its genericity makes it more cumbersome than CUDA to use in applications. Instead, OpenCL is more commonly used to create higher-level development frameworks or in commercial applications that need to run on a wide variety of hardware platforms.

Although being more generic than CUDA, it provides higher abstractions in some areas like memory management. For example, memory is allocated in a *context* instead of a device. A context can be bound to several devices and it is the OpenCL runtime who decides in which physical memory it is allocated. Due to the different capabilities of the devices that support OpenCL, mechanisms like remote memory accesses are not supported either. Since the proposed techniques heavily rely on these mechanisms, we use CUDA in the rest of the thesis.

Chapter 3

State of the Art

Programmability of multi-GPU systems can be tackled at different levels, ranging from system support to memory management and transparent computation decomposition and distribution.

3.1 Automatic CPU/GPU memory coherence

GPUs use different memories or separate memory subsystems than CPUs. This provides a performance advantage but, on the other hand, programmers are forced to use several copies of data and keep them coherent. Some solutions have been proposed to automatize the CPU/GPU memory coherence.

The ADSM model proposed by Gelado et al. [45] presents a Unified Virtual Address Space (UVAS) which is shared by CPU and GPU. ADSM allows programmers to declare objects once and use them both in GPU and CPU functions with no need for explicit memory transfers at all. The ADSM runtime transparently creates the needed copies of the objects and makes sure that they are coherent at consistency points. It uses acquire/release consistency at kernel call boundaries. ADSM is implemented in the GMAC user-level library [10] that provides eager GPU memory update to transparently overlap CPU computation (e.g., data initialization) and data transfers. GMAC uses the memory protection mechanism of modern CPUs to detect with pages are accessed in the host code. However the work in these thesis targets multi-GPU systems while ADSM is restricted to a single GPU. In Chapter 5, we present the HPE model that extends ADSM to multi-GPU systems.

While ADSM greatly simplifies GPU development, programmers still need to use a specialized allocation/free functions for data that is shared between CPU and GPU. Jablin et al. propose compiler analysis in [52] to

automatically determine which data structures are shared. They also add support for stack allocations and global variables, while GMAC is limited to heap allocations. The runtime system inspects the parameters at kernel call boundaries and keeps track of the location of data to determine when data transfers are required. This approach prevents the utilization of complex data types that use pointers, as they should be recursively inspected. Jablin et al. extend this work in [51] by replacing the parameter inspection with the memory protecting mechanism like the one used in GMAC. These works, like ADSM, target systems with a single GPU, only.

Some higher-level languages have been proposed to simplify memory management in GPU programs. C++ AMP [3] provides a multi-dimensional array data type to represent the data that is used in the GPU kernels. Objects created from this type can be used in the CPU code, too. Using this type instead of raw pointers allows the compiler to detect the memory objects passed to each GPU kernel and insert the appropriate coherence annotations for the runtime system. The solution proposed in Chapter 7 uses the same approach of using a special data type, but the proposal in Chapter 5 works for any type of memory allocation.

3.2 System support and GPU virtualization

GPUs are commonly presented as isolated compute-only devices that cannot interact with other devices (e.g., network interfaces or disks). Moreover GPU programs are not well integrated with many of the mechanisms offered by Operating Systems such as inter-process communication and memory paging.

Silberstein et al. propose GPUfs in [82], a POSIX-like API for GPU programs that makes the file system directly accessible to GPU code. The API is designed to exploit structured data parallelism such that threads in the same warp cooperate to perform read/write operations. GPUfs is mostly implemented as a library that works with the host OS on the CPU to coordinate the file system's namespace and data. The GPU sends file operation requests to the CPU while the kernel is running, by using a shared communication buffer. A kernel module component enables caching both in the CPU and GPU memories by distributing the buffer cache in the OS. Different GPUs and CPUs can concurrently work on the same file and changes are consolidated using diffing[1] at synchronization points.

Kim et al. present GPUnet in [58], a native networking layer that provides a socket abstraction and high-level networking APIs to GPU programs.

[1]Diffing is a technique that compares a memory buffer with its original copy to find the values that have changed.

GPUnet removes the need for programmers to coordinate NIC, CPU and GPU in order to transfer data that resides on the GPU memory. Further, GPU kernels can trigger network transfers and, therefore, programs no longer need to wait for kernel finalization in order to transfer data. GPUnet takes advantage of GPUDirect, enabling direct communication between NIC buffers and GPU memories, and avoiding intermediate copies to host memory. Optimized paths are implemented for communication between sockets of GPUs that reside in the same node (and do not require using the NIC).

Rossbach et al. [78] propose a new abstraction called PTask for processes that run on the accelerator and the addition of ports and channels to represent the communication graph among regular processes and PTasks. Using this scheme, unnecessary memory transfers among CPU and GPU memories can be avoided since the placement of memory objects is known to the system runtime. Moreover, more intelligent scheduling policies can be implemented by taking advantage of the features provided by the accelerators in the system (e.g., concurrent GPU execution and memory transfers).

Duato et al. propose the rCUDA middleware. rCUDA virtualizes the CUDA-RT API and implements a client/server architecture to execute applications on remote GPUs. In [40], authors use rCUDA to manage all the GPUs in a cluster. This allows applications to use remote GPUs in a similar way as if they were local GPUs. It also enables cluster-level scheduling, which leads to higher GPU utilization. In [39], rCUDA allows programs in guest Virtual Machines to access the GPUs on the physical machine. Compared to our work, rCUDA is limited to and requires a complete implementation of the CUDA-RT API, while the HPE model proposal in Chapter 5 can be implemented in different programming models (versions exist for CUDA and OpenCL).

Similarly to rCUDA, Shi et al. propose vCUDA in [80] to implement GPU sharing for programs running in guest Virtual Machines. It provides a virtual GPU view to each application running on the node, though multiple applications actually share a single physical GPU. Instead of sharing among applications, our solution in Chapter 7 aggregates all the GPU resources into a single virtual GPU to transparently scale the performance of applications.

Virtualization proposals for OpenCL-based clusters also exist. Barak et al. introduce VCL in [25], a solution based on MOSIX that exposes all the GPUs in the cluster to standard OpenCL programs. Xiao et al. present VOCL in [98, 99]. Besides the transparent access to remote GPUs, VOCL is able to perform live migration of *virtual GPUs* between different physical GPUs in a cluster. Like rCUDA, these solutions are limited to applications that use the OpenCL API.

Kato et al. propose in [55] a more generic GPU virtualization solution

by implementing GPU-aware resource management in the OS, called Gdev. Gdev integrates virtual memory management in the OS kernel, thus allowing to implement mechanisms such as shared memory between processes. It also implements preliminary support for paging although GPUs do not provide support for restartable page faults (needed to implement on demand paging). Instead, when a process requests more memory than is available on the GPU, Gdev evicts data from other processes using the GPU and moves it to host memory.

HSA [16] (i.e., Heterogeneous System Architecture) is a industry standard for heterogeneous systems that defines a set of features that need to be supported by devices. Many of the members of the HSA Foundation are major silicon vendors such as AMD, ARM and Samsung. One of the main objectives of HSA is to completely remove the need for programmers to explicitly manage all the memories in the system. Similarly to our proposal in Chapter 5, HSA provides a single flat virtual address space in which all the memory can be accessed by any processor in the system. HSA also requires support for context switching and paging to implement system-wide policies. Currently, only a few devices have support for HSA and support in GPUs is limited to integrated designs such as the ones in AMD APUs. While HSA theoretically supports any high-level programming language, current HSA-compliant GPUs use OpenCL.

3.3 Automatic multi-GPU execution

3.3.1 Compiler-based transparent multi-GPU execution

Kim et al. [57] introduce an OpenCL framework that combines multiple GPUs and treats them as a single compute device. Thus, when a kernel is launched, they decompose computation and data across the GPUs in the system. The computation grid of the kernel is decomposed into uniform partitions that are distributed across GPUs. In order to perform data decomposition, they compute the array range accessed by each computation partition by performing a sampling run of the kernel on the CPU. This step is only performed if array references are affine transformations of the thread and block identifiers, which is determined using compiler analysis; otherwise the whole array is replicated in all GPU memories. This solution does not detect the dimensionality of the arrays used in the GPU kernels. Therefore, even in cases where data can be decomposed, any array decomposition not performed on its highest-order dimension will produce tiles whose memory address ranges overlap, replicating big portions of the array in all memories.

The main problem of using replication is that it reduces the size of the problems that can be handled. Another drawback is that replicated array regions that are potentially modified from different computation partitions need to be merged after every kernel execution. This step is executed on the CPU, thus increasing CPU↔GPU traffic and imposing a large overhead in many computations. Furthermore, atomic operations do not on replicated data, either.

Lee et al. [61] propose SKMD, which extends the same idea to heterogeneous systems with CPUs and GPUs. SKMD performs a profile of the different devices in the system to distribute the computation partitions according to their capabilities. They do not use the sampling runs on the CPU and solely rely on the compiler to detect the array region accessed by each partition, which leads to replication in more cases than in [57]. On the other hand, they generate a merge kernel (based on the original kernel code) for replicated data, that is more efficient. However, this step is still performed on the CPU, requiring all the replicated copies to be transferred to host memory (which is the part of the merge step that incurs the most overhead).

The solution proposed in Chapter 7 exploits the support for remote access across GPU memories thus avoiding, in most cases, data replication and merge operations that are required in these previous works.

3.3.2 Compiler-based multi-GPU code generation

Lee et al. [63, 62] provide automatic GPU code generation called OpenMPC, which relies on standard OpenMP annotations for the host code. It detects the variables accessed in each loop (that can be explicitly specified through clauses or implicitly), the access type, and implicit synchronization points. A second phase divides parallel regions at synchronization points to enforce correctness (there are no kernel-wide synchronization primitives in CUDA). The next phase transforms the CPU-oriented kernel into a CUDA kernel and performs CUDA-specific optimizations. The compiler inserts calls for GPU memory allocation and data transfers between CPU and GPU memories. Sabne et al. [79] extend OpenMPC to support out of core computations and multi-GPU execution. It also provides communication and computation overlap. The achieved speedups are competitive with hand-written CUDA versions for a single GPU but they do not scale when several GPUs are used.

PGI proposes custom directives for Fortran programs to automatically generate optimized code for accelerators in [97]. Clauses are included to explicitly specify data transfers to the GPU memory. The compiler performs

strip-mining[2] on the program loops to generate inner loops and assign the different loops to block-level and thread-level parallelism offered in CUDA. These directives have been later extended and published as the OpenACC standard [75] which is supported by many hardware and systems vendors. OpenACC, however, requires programmers to manually decompose computations so that they can be executed on multiple GPUs.

Our solution presented in Chapter 7 uses compiler analysis to automatically determine how computation and data can be decomposed and distributed, achieving linear speedups.

3.3.3 Multi-GPU libraries

Using libraries is a simple way for programs to transparently be able to run efficiently on different computer architectures. Programmers only need to express computations in terms of calls to library functions, with no knowledge of the characteristics of the hardware. Libraries can implement optimized versions of these computations for different accelerator architectures and system topologies.

MAGMA [93] is a widely-used linear algebra library with support for large systems. It implements heterogeneous execution of computation kernels that uses both CPUs and GPUs. Unfortunately, only a subset of the provided functions is able to exploit several GPUs.

Nukada et al. [73] present a 3D FFT library for CUDA. It uses autotuning to generate CUDA kernels that are optimized for different transform sizes.

Babich et al. [24] propose QUDA, a library for numerical lattice quantum chromodynamics (LQCD) calculations. Thanks to a multi-dimensional decomposition of the problem, they are able to scale the performance to up to 256 GPUs.

ArrayFire [17] is a C/C++/Fortran library that provides abstractions for multidimensional arrays and a number of libraries that use them (e.g., data analysis, linear algebra, image and signal processing). However, the utilization of these arrays is limited to the functions offered in their libraries and cannot be used in custom user-defined kernels.

In the latest versions of CUDA, NVIDIA offers cuBLAS-XT [8], an extension of their cuBLAS [7] library that is able to decompose and spread work across several GPUs for a set of Level 3 BLAS calls. Moreover, they also provide NVBLAS [11], which provides automatic multi-GPU acceleration for applications that use regular BLAS calls. NVBLAS builds on cuBLAS-XT

[2]Strip-mining reshapes a multi-dimensional space to create additional dimensions from the elements of one of the original dimensions. It is commonly used to perform blocking.

and intercepts calls from the application to regular BLAS libraries, and replacing them with GPU calls, transparently.

The main problem of the library approach is that only the computations provided in the library are able to transparently use multiple GPUs.

3.4 Languages for multi-GPU execution

Several languages (that mainly use the PGAS model) have been proposed or extended to ease the exploitation of multiple accelerators in large systems. These PGAS (Partitioned Global Address Space) languages require a complete rewrite of the program and are not typically used to port existing codebases.

Universal Parallel C (UPC) has been extended in [100] to transparently access data allocated in GPUs. UPC uses a Hybrid PGAS in which each CPU thread is bound to one shared segment, that can be either in host memory or in GPU memory, but not both. Each thread is bound to the same memory segment during its lifetime.

GlobalArrays [70] (GA) is a library that allows execution of computations across a distributed system using an array-like interface. GA-GPU [91] extends GA to GPUs, and differentiates explicit and implicit utilization models. The implicit model is offered to programmers through a pre-defined set of data-parallel primitives such as linear algebra operations (e.g., `GA_Dot`). It is preferred to the explicit model because its computations are encapsulated and the runtime knows how to distribute computation data and which portion of the array is used in any GPU. On the other hand, launching a kernel for each of the operations imposes an overhead, compared to using a custom kernel that contains all the operations. Custom kernels from programmers (i.e., the explicit model) need to honor the memory model in GA. This memory model allows memory accesses ordering and system-wide memory fences and, hence, it does not fit into bulk synchronous SPMD programming models such as CUDA or OpenCL.

X10 [34] provides a Java-like syntax that is able to generate code versions for different targets (e.g., multicore, GPU). X10 presents an *Asynchronous* PGAS in which the memories in the system are abstracted as *places*. Each computing device is bound to a place, although they can manage (copy to/from) objects in remote places. Since CPUs and GPUs are bound to different places, explicit memory transfers are required between CPU and GPU objects. To improve performance, copies and kernel execution can be spawned asynchronously and managed through explicit synchronization primitives. Habanero [33] builds on top of X10 and adds abstractions to better integrate

stream parallelism with task parallelism.

Lime [41] extends Java with several constructs to program heterogeneous architectures. The Lime compiler emits OpenCL code that can target different types of accelerators. Lime includes the *task* operator to create coarse-grain tasks, and *map* and *reduce* primitives for fine-grained data parallelism. The communication between tasks is expressed through the *connect* operator, which allows the compiler and runtime system to optimize I/O and synchronization without programmer intervention.

Chapel [36] is a programming language developed by Cray designed to provide performance scalability in large systems. It provides abstractions for data parallelism, task, concurrency and nested parallelism. Chapel supports a global view of data structures, so that programmers can easily perform computations on distributed data. On the other hand, it allows to reason about data and computation placement, thus enabling performance tuning. Sidelnik et al. [81] add support for GPUs in Chapel. They implement GPU code generation by adding a new clause that lets programmers define the dimensionality and size of problem and data. Two data management models are offered: implicit (i.e., fully automated) and explicit for hand tuning. However, it only generates code for single-GPU execution.

Sequoia [42] tries to address the problem of programming systems with different memory topologies/hierarchies. Programs written for Sequoia are composed of two parts: (a) an algorithmic representation of the computation using a C-like programming language that decomposes data structures and defines how to map the computation on them; and (b) a mapping of the algorithm to the specific system using a declarative language. Legion [94] is a generalization of Sequoia that exposes abstractions to declare the properties of program data. Programmers dynamically organize data into *regions* and define how regions are accessed (i.e., access privileges) and computed. Using this information, Legion tries to implicitly extract parallelism to distribute the computation across all the computing devices and optimize data movement.

Task-based programming models

Ayguadé et al. present the GPU Superscalar (GPUSs) programming model and runtime in [23]. GPUSs relies on annotations to host functions and CUDA kernels, processed by a source-to-source compiler, to create a data dependency graph. These annotations are based on the OpenMP task pragmas, that define what data is used in each task, and the type of access. A runtime is in charge of scheduling kernel execution, allocating memory and performing data copies among the GPUs in the system.

Augonnet et al. present StarPU [22], a runtime dependency tracker and scheduler for heterogeneous systems. Programmers write tasks in the form of *codelets* and define the dependences among tasks. Contrary to GPUSs, StarPU allows programmers to define arbitrary dependences between tasks (using task identifiers or tags). The runtime dynamically chooses the version of the codelet to be executed depending on the load of the different processors in the system.

Although both solutions free programmers from explicitly allocating GPU memory, performing memory transfers and scheduling kernel execution, they still have to manually decompose computations into tasks and define task inputs and outputs.

Chapter 4

Guiding Example: Reverse Time Migration

In this thesis, as a guiding example, we decided to take an important application in the oil and gas industry and port it to multi-GPU systems. This work has been done in collaboration with the Repsol oil and gas company. In this document, we also compare the performance of our implementation against implementations for general purpose CPUs, the Cell B.E., and FPGAs.

4.1 The Reverse Time Migration computation

RTM's major strength, compared to previous solutions [86], is the capability of showing the bottom of salt bodies at several kilometers (∼6 km) beneath the earth's surface. The USA Mineral Management Service (MMS) reports [26] can help understand the impact of RTM. The oil reserves of the Mexican Gulf under the salt layer are approximately 5×10^{10} barrels. Moreover, the reserves in both Atlantic coasts, Africa and South America, are also under a similar salt structure. RTM is the key application to localize these reserves. RTM is the method that produces the best subsurface images; however, its computational cost (at least one order of magnitude higher than others) hinders its adoption in daily industry work. Fortunately, accelerators such as GPUs open the door for a high performance implementation of RTM. The used RTM implementation is based on an explicit 3D high-order Finite Difference numerical scheme for the wave equation, including absorbing boundary conditions. Therefore, other simulations based on the same numerical scheme will benefit from some of our conclusions. For example, magneto-hydrodynamics in astrophysics, atmospheric mesoscale model-

ing and some DNS turbulent flow simulations.

4.1.1 Seismic imaging

Seismic imaging tries to generate images of the terrain in order to see the geological structures. This is the basic tool of oil industry to identify possible reservoir locations. Moreover, this technology is needed for several geological activities, for example, the CO_2 sequestration. The main technique for seismic imaging generates acoustic waves and records the earth's response at some distance from the source. Like in medical imaging, using some signal processing algorithm on the recorded data, RTM rebuilds the properties of the propagation media, i.e., the local earth structure. The propagation of waves in the earth is a complex phenomenon that requires a sophisticated wave equation to be modeled accurately. Unfortunately, the amount of information about local earth's properties is very little, and simplified models for wave propagation are used. The simplest model assumes that wave propagation is modeled by the isotropic acoustic wave equation:

$$\frac{d^2p(r,t)}{dt^2} + c(r)^2\nabla^2 p(r,t) = f(r,t) \tag{4.1}$$

where $c(r)$ is the sound speed at each terrain point r, $p(r,t)$ is the pressure wave value, and $f(r,t)$ is the input wave. This simple model uses a single parameter $c(r)$ to model the terrain properties. Using this acoustic model it is not possible to reproduce the complete wave form of the received wave. However, this model is good enough to reproduce the arrival time of the received waves. Even considering that the amplitude information is lost by the acoustic model, the phase information is preserved.

RTM solves an inverse problem based on Equation (4.1). The inverse problem solution is decomposed in two functions:

1. Tomography is the function that builds an approximate velocity model $c(r)$, using a terrain image and the empirical data.

2. Migration is the function that builds a terrain image, using a velocity model and the empirical data.

RTM algorithm

Due to the fact that the acoustic shots (medium perturbation) are introduced in different moments, they can be processed independently. The most external loop of RTM sweeps all shots. This embarrassingly parallel loop can be distributed in a cluster or a grid of computers. The number of

shots ranges from 10^5 to 10^7, depending on the size of the area to be analyzed. In this thesis, we focus on the RTM algorithm needed to process one shot. Listing 4.1.1 shows the pseudocode of this algorithm. RTM is based on solving the wave equation Equation (4.1) two times. First, using as right-hand side the input shot (InsertShot), and second, using as right-hand side the receivers traces. Wave propagation is performed by using Finite-Difference Time-Domain (ComputeWaveField) and uses absorving boundary conditions (ComputeABC) to avoid wave reflections from the edge of the computational domain. Then, the two computed wave fields are correlated (Correlate) at each point to obtain the image. In order to be able to correlate the wave fields, snapshots of the first one are saved in external storage during the forward propagation phase (StoreForward), and read later in the backward propagation phase (ReadForward).

```
procedure Forward(V, SL, T)
V: velocity model
SL: shot location
T: number of steps to be simulated

    P ← InitGrid
    for all t ∈ T do
        P ← ComputeWaveField(P)
        P ← InsertShot(P, SL)
        P ← ComputeABC(P)
        StoreForward(P)
    end for
end procedure
```

```
procedure Backward(V, R, T)
V: velocity model
R: traces from the on-site receivers
T: number of steps to be simulated

Returns C: image
    P ← InitGrid
    C ← InitGrid
    for all t ∈ T do
        P ← ComputeWaveField(P)
        for all r ∈ R do
            P ← InsertShot(P, r_t)
        end for
        P ← ComputeABC(P)
        F ← ReadForward(t)
        C ← Correlate(P,F,C)
    end for
end procedure
```

Listing 4.1: Pseudocode of the RTM algorithm.

4.2 RTM base implementation

RTM discretizes wave fields into a regular 3D grid in which each element represents the scalar value of the wave in a region of the space. The wave propagates across the cells of the volume using FDTD (i.e., Finite-Difference Time-Domain), which is implemented using a 3D stencil and integration in time computations that are executed for a number of time steps. Both computations can be merged and we refer to them as a single computational kernel in the rest of the chapter. An stencil computation uses the central

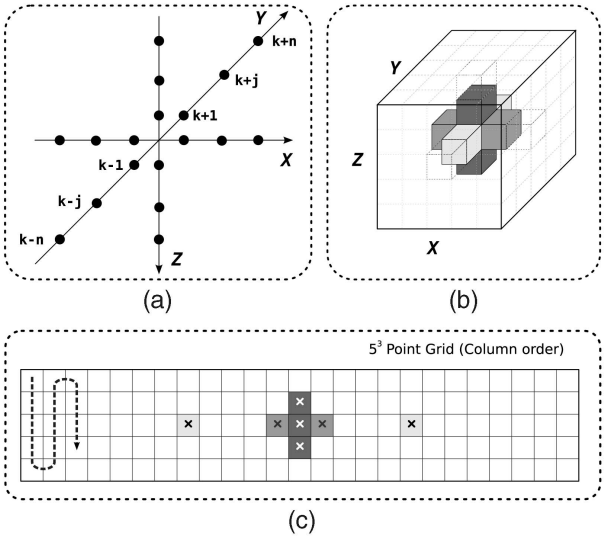


Figure 4.1: (a) The 3D stencil computation pattern. (b) 3D 7-point stencil within a volume. (c) Memory access pattern of a 3D 7-point stencil.

value and the k neighboring elements in each dimension of the volume to compute each output cell (Figure 4.1 a and Figure 4.1 b). This computation is the most computationally demanding kernel in RTM. The wave is generated by exciting the elements where the ship performed the real shot in the forward propagation phase, and introducing the values gathered by the receivers in the backward propagation phase. The absorving boundary conditions are implemented by adding a few elements at the boundaries of each dimension of the volume. The cells in this *halo regions* are read in the stencil kernel but they are written by a different kernel that generates values that absorve the energy of the wave.
RTM is a very demanding application that poses challenges to developers in several areas. We describe these problems in the next subsections.

4.2.1 Memory

RTM memory consumption is related to the frequency at which the migration is done. Frequencies over 20-30 Hz may require the utilization of several GiB (>10 GB) of memory for computing a single time step for a single shot. While each shot can be computed independently, the amount of required memory can exceed the available in a single computational node, forcing a domain decomposition technique to process one shot.
The 3D stencil computation step only needs accessing the data of the wave

field on the current time step. However, the velocity model and the contents of the two previous time steps of the wave field are also required for the time integration and the absorbing boundary conditions steps. Additionally, a fifth volume is necessary to store the image illumination. The size in memory of a wave field volume depends on the dimensions of the field, but an additional ghost area is used in each dimension to calculate the 3D stencil. The velocity model and the illumination volumes, on the other hand, do not need the ghost area.
The memory requirements during the backward propagation phase are very similar. The illumination volume used in the forward phase is replaced by another one used to compute the final image by performing the correlation between the forward and backward wave fields. Additionally, it also uses the information of the receivers. Each receiver has a value for each simulated time step.

4.2.2 Input/Output

The I/O problem can be divided into three categories: data size, storage limitations, and concurrency. First, looking for high accuracy, the spatial discretization may produce a huge computational domain. Second, the time discretization may imply large number of time steps (typically $> 10^4$).
RTM implementations store the whole computational domain regarding the number of time steps in the forward propagation phase. Thus, the storage requirements for a single shot (>1 TB) largely exceed the capacity of the RAM memory and must, therefore, be transferred to external storage (e.g., hard disk drive). In order to avoid that RTM becomes an I/O bound application, it is mandatory to overlap computation and I/O using asynchronous libraries or multiple threads. Additionally, data is compressed before being transferred to disk in order to reduce disk transfer communication times and storage requirements. Furthermore, the correlation can be performed every n steps at the expense of image quality (we call this *stack rate*).

4.2.3 Computation

RTM presents two main characteristics regarding computation:

1. **Low computation versus memory access (c/ma) ratio**: Many neighbor points are accessed to compute just one central point and, even worse, many of the accessed points are not reused when the next central point is computed. This effect is called low data locality ratio. For instance, the generic stencil structure in Figure 4.1 a defines a

$(3 \times (n \times 2)) + 1$ stencil access pattern. If $n = 4$, then 25 points are required to compute just one central point, then the c/ma ratio is 0.04, which is far from the ideal c/ma 1 ratio. To tackle this problem, strategies that increase data reuse must be deployed.

2. **Large data parallelism**: The stencil computation performs the same operations in every single cell of the volume (excluding the halo regions). The boundary conditions' kernel also exhibits a lot of data parallelism. Further, the insertion of the information of the receivers can also be performed in parallel. Therefore, vectorial functional units present in modern processors and accelerators can be exploited to achieve the maximum performance.

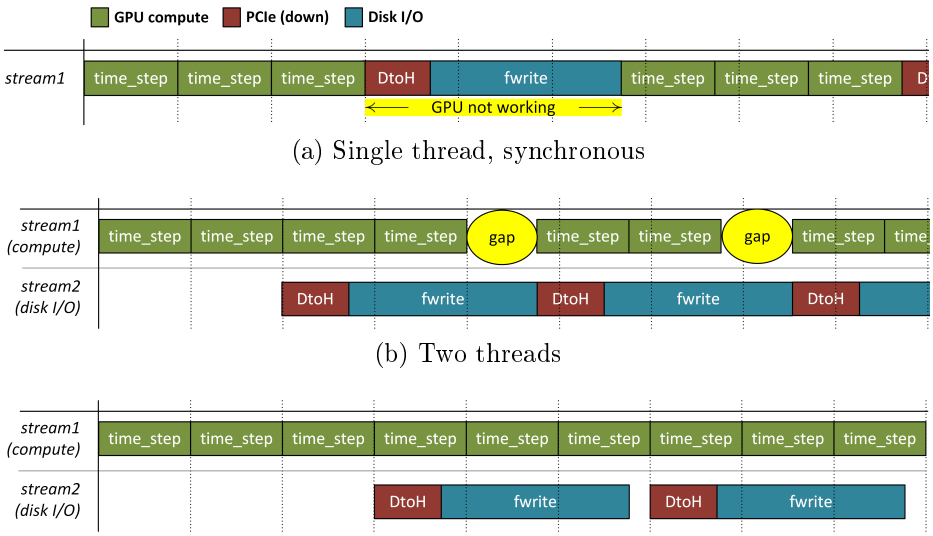
4.3 Porting RTM to GPUs

A GPU executes computational kernels written in SPMD languages such as CUDA. Data must be transferred to the GPU's GDDR memory before the kernel that uses it is executed. GPU's performance is hugely penalized by frequent and large memory transfers through the PCIe bus. Therefore, we have implemented all the RTM computations as GPU kernels that operate on data which always resides in the GPU's GDDR memory. This limits the size of the problem that can be handled on a single GPU. The host code only orchestrates the execution environment, the kernels' invocations, and the I/O transfers to/from disk whenever they are necessary.

4.3.1 Data I/O schemes

In the forward propagation stage, at every stack step, the wave field is compressed and transferred to disk. Since GPUs have their own memory, an additional transfer between the GDDR and the host memory is necessary, as the data cannot be directly transferred to an external device (which would require extra support from the hardware). The transfers through the PCIe are much shorter since it provides a bandwidth (up to 16 GBs) much higher than that of disks. In the backward propagation stage, the order of the transfer steps is the opposite. Using a completely synchronous model lowers the utilization of the GPU, as it is blocked until disk I/O is completed (Figure 4.2a). In order to hide the costs of these memory transfers, we use the following scheme:

- Asynchronous PCIe transfers. Memory transfers in CUDA are synchronous and blocking by default. Blocking the CPU thread while the



(a) Single thread, synchronous

(b) Two threads

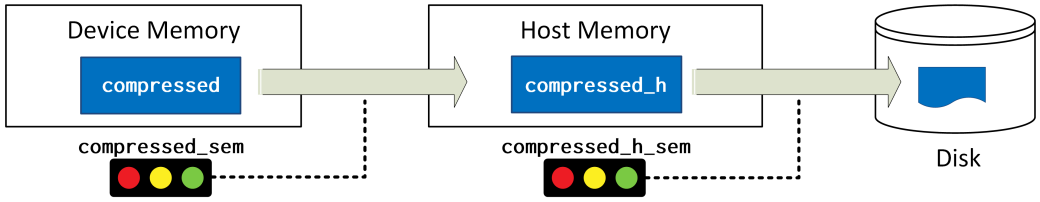
(c) Two threads, perfect overlap

Figure 4.2: Execution timelines of the GPU implementation of RTM for different data I/O schemes.

compressed wave field is transferred to host memory prevents it from keep invoking kernels for the next simulation time steps. In order to avoid this problem, we use different CUDA *streams* for data transfers and kernel execution (Figure 4.3). Moreover all data transfers performed on separate streams if the destination/source memory buffer in host memory is *pinned*. This allows the OS to program the DMA transfer through PCIe using the buffer provided by the programmer. Otherwise additional copies to internal pinned buffers allocated by the OS are performed.

- Dedicated buffer for the compressed wave field. This comes at the cost of extra memory utilization (that depends on the used compression ratio). However, computation can thus continue on the GPU as soon as the wave field has been compressed. The costs are hidden as long as the transfers take less than the execution time steps between different stack steps.

- Separate thread for PCIe + I/O. While asynchronous PCIe transfers allow to overlap computation and communication, it forces programmers to query the state of the transfer in each iteration of the simulation. Further, if the transfer finishes while the CPU thread is blocked waiting for the commands in a stream (e.g., kernel calls) to finish, it is not noticed until later (thus inserting a buble in which the disk I/O




```
sem_init(compressed_sem, 0, 1);
sem_init(compressed_h_sem, 0, 0);
…
if (step % stack_steps == 0) {
  sem_wait(&compressed_sem);
  compress<<<kernel_conf,streamComp>>>();
  cudaStreamSynchronize(streamComp);
  sem_post(&compressed_h_sem);
}
...
```

```
...
if (step % stack_steps == 0) {
  sem_wait(&compressed_h_sem);
  cudaMemcpyAsync(compressed_h,
                  compressed,
                  cudaMemcpyDtoH,streamCopy);
  cudaStreamSynchronize(streamCopy);
  sem_post(&compressed_sem);
  fwrite(compressed, 1, bytes, file);
}
...
```

Figure 4.3: Synchronization scheme used to overlap I/O and communication in the GPU implementation of RTM.

could have started). Thus, we use a separate CPU thread that executes all the PCIe and disk operations on a separate thread. This also eliminates the necessity for asynchronous disk I/O operations. On the other hand, this requires the utilization of inter-thread synchronization primitives (see Figure 4.3), which adds extra complexity to the code. One semaphore is used to notify the CPU thread that the compressed buffer is ready (`compressed_h_sem`). A second sempahore is used to notify that the copy has finished and the buffer can be overwritten (`compressed_sem`). This scheme reduces the amount of time that the GPU is not used but, depending on the capabilities of the I/O, bubbles can appear (a stack rate of 2 produces bubbles in Figure 4.2b) and higher stack rate values might be required in order to completely hide the costs of the transfers (Figure 4.2c shows that increasing the stack rate to 3 removes the bubbles), at the cost of lower accuracy.

4.3.2 GPU kernels

Our port targets the NVIDIA Tesla C1060 GPU, whose characteristics can be seen in Table 4.1.

3D Stencil

Accesses to global memory are very expensive and, while modern GPUs provide caches, they are too small to store all the accessed elements in the

GPU test system	
Processors	Xeon E5460
Clock frequiency	2.4 GHz
Sockets	1
Cores per socket	4
GPU	C1060
SMs	30
SIMD registers	16K (SM)
SIMD width	1024 bit
GPU memory bus width	512 bit
GPU memory	4 GB
GPU shared memory	16 KB (SM)
Host memory	8 GB
Memory standard	DDR3
Interconnect	PCI Express 2.0 ×16
Peak throughput	933 GFlops

Table 4.1: Hardware configuration used for the evaluation of the GPU implementation of RTM.

stencil computation. Therefore, optimizing global memory bandwidth usage is a must in order to obtain good performance from the GPU. A k-order stencil computation calculates the value of each point by using the k neighbor elements in each direction. That is, $3k + 1$ reads per calculated element. This number is referred to as read redundancy by Micikevicius in [68]. The same concept is also applied to writes. The sum of them is called overall redundancy. We used this metric to analyze global memory accesses and improve memory bandwidth usage.

In this kernel, we use a 2D sliding window approach [68] in which each thread computes the same element in every plane of the volume, but all threads within a thread block work on the same plane at a time. Thus, shared memory is used to store an $(n + k) \times (m + k)$ tile which holds elements of the z and x dimensions of a plane of the wave field. Each thread loads an element into the shared memory, and some threads also load halo elements needed to compute the points in the tile boundaries. Since threads load elements that are consecutive in memory, they can be coalesced in order to maximize the global memory bandwidth. Accesses to neighbor elements for these dimensions are then fetched from shared memory, thus avoiding requests to the regular memory hierarchy. The kernel iterates on the y dimension; thus, each thread computes a column along this dimension. Neighboring elements in the y dimension are stored in k registers (private to each thread) which behave like a queue: every time a tile is done, the oldest element is popped out and a new element is pushed.

The bigger the thread block, the lower the read redundancy as the ratio of elements in the halo regions of the tile is smaller. On the other hand, using large thread blocks can reduce the concurrency in the SMs because a single thread block can monopolize all the resources. A multiple of 32 in the x dimension of the thread block is required in order to have memory coalescing. Given this constraint, we experimentally determine that 32×4 is the thread block configuration that provides the best performance with a read redundancy of 3.25 (for $k = 4$) and SM occupancy of 100%. Our implementation also has a write redundancy of 1.

Absorving Boundary Conditions

We have implemented a different kernel for each face of the 3D wave field as they require different memory access patterns. The used approach is the same as the one used for the stencil computation: there is a front of threads that traverses the points that belong to the ABC area and updates them accordingly. In this case, the dimension which is traversed depends on the face that is being computed. This makes memory coalescing not possible when computing two of the faces since none of the two dimensions of these faces is consecutive in memory. The slowdown for these two faces can be up to 6× compared to any of the other four faces. However, the execution time of the boundary condition kernels is much lower than the 3D stencil kernel.

Shot insertion

This computation is very simple and could be run in the host. However, since the wave fields reside in the GPU memory, additional memory transfers should be performed in order to keep the data coherent. Thus, we have implemented a simple kernel that uses a single thread in the GPU.
The backward phase of the RTM algorithm requires exciting the medium with the data gathered by the receivers in the field. The algorithm is the same one used for the shot insertion so the code has been reused. Furthermore, there can be up to thousands of receivers. Thus, we exploit the GPU parallelism and create one thread per receiver to calculate all of them in parallel. Receiver's data is stored in such a way that contiguous threads read contiguous elements in memory.

Compression

We implement a lossy compression algorithm that discretizes the floating point values within a range of integer values. Each thread block in the kernel compresses/decompresses a region of the wave field. In order to improve the

Blade SGI Altix XE320	
Processors	Xeon E5460
Clock frequiency	2.5 GHz
SIMD registers	N/A
SIMD width	128 bit
Sockets	2
Cores per socket	4
Memory per blade	8 GB
Memory standard	DDR3
Peak throughput	80 GFlops

Table 4.2: Hardware configuration used for the evaluation of the CPU implementation of RTM.

accuracy, we compute and store the maximum and minimum values within each region, and the integer values represent the offset within that range. Thus, the compression kernel first performs a reduction step to find the maximum and minimum values in the region. Threads in each thread block work on contiguous elements of the Z dimension of the grid, which are contiguous in memory and, hence, provide memory coalescing.

4.4 RTM implementation on other architectures

We compare the performance of our GPU implementation with previous optimized implementations on general purpose CPUs, the Cell B.E. processor, and the Convey HC-1 FPGA system.

4.4.1 General purpose CPUs

We compare against the CPU implementation in [21], that uses the hardware configuration in Table 4.2.
The stencil memory access pattern is a main concern when designing the RTM kernel code [54], because it is strongly dependent on the memory hierarchy structure of the target architecture. Besides, due to the reduced size of the L1, L2, or L3 caches, blocking techniques must be applied to efficiently distribute the data among them [96], at least for classical multicore platforms. To apply blocking, in particular the Rivera [77] strategy, the original 3D space is divided in slices, where the X axis of each has a size that optimally fits the cache hierarchy
This implementation exploits all the forms of parallelism provided by the

QS22	
Processors	Cell B.E.
Clock frequiency	3.2 GHz
SIMD registers	128 (SPE)
SIMD width	128 bit
LS size	256 KB (SPE)
Sockets	2
Cores per socket	1 PPE + 8 SPEs
Memory per blade	8-32 GB
Memory standard	XDR
Peak throughput	205 GFlops

Table 4.3: Hardware configuration used for the evaluation of the Cell B.E. implementation of RTM.

architecture; the thread-level parallelism provided by the multiple cores, and the data-level parallelism provided by the SIMD instruction set. Eight independent threads are used with a parallelization strategy that follows the blocking. Each core processes its assigned set of slices independently. Since each core has its own L2 cache, interference among threads is minimal.

4.4.2 Cell B.E

As a reference Cell B.E. platform, we consider the IBM Blade-Center QS22 blade (see Table 4.3 for the full specifications), which has two sockets. Each Cell B.E. processor on a QS22 blade contains a general-purpose 64-bit PowerPC-type (i.e., PPE) with cache memories, and eight SPEs with software-based scratchpad memories called Local Stores (LS). The PPE and the SPEs both have a (slightly different) 128-bit-wide SIMD instruction set, which allows, for example, to simultaneously process four single-precision floating-point operands. Contrary to GPUs that use regular `load-store` instructions to move data to *shared memory*, explicit DMA transfers must be used in order to transfer data from the memory to the LS.
Blocking is used to evenly distribute work and data among the SPEs' local stores. Similarly to our GPU implementation, the 3D space is split in the X direction, so that each subvolume is given to one SPE to be processed (see Figure 4.4). Y is the traversing direction while each subvolume is processed. This is particularly important in the Cell B.E. processor, due to the limited size of the SPEs local stores. Thus, a streaming of Z-X planes is constantly providing the SPEs with the required data to compute.

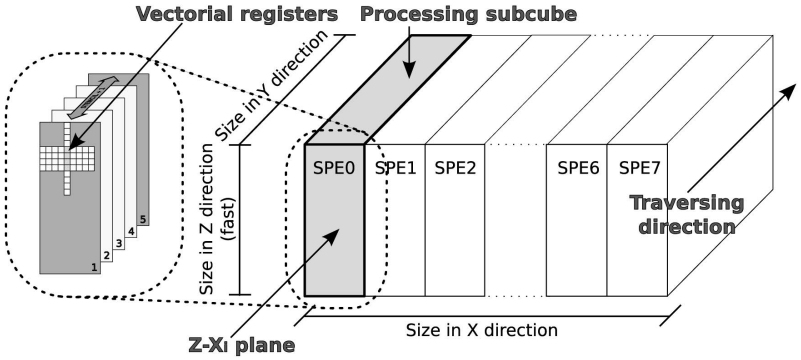
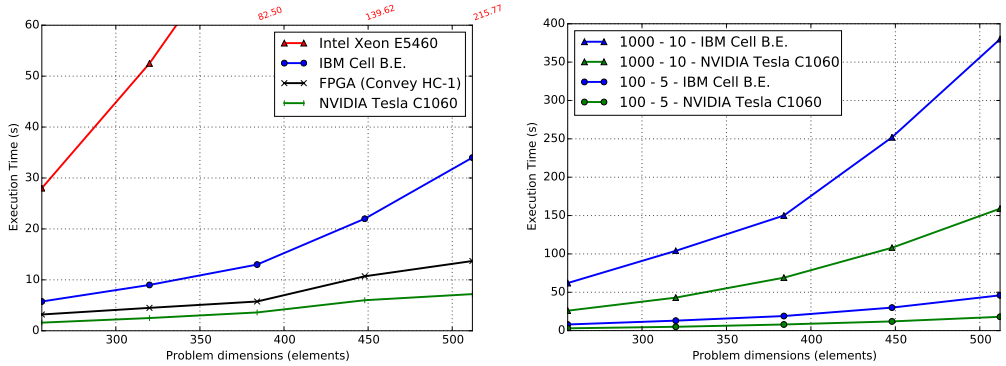


Figure 4.4: Data access and vectorization pattern for the Cell B.E.

Convey HC-1 FPGA system	
Processors	Xeon E5345
Clock frequiency	2.33 GHz
Sockets	2×1
Cores per socket	4
FPGA	4× Virtex 5 LX330
Host memory	128 GB
Memory standard	DDR3
Interconnect	PCI Express 2.0 ×16
Peak throughput	760 GFlops

Table 4.4: Hardware configuration used for the evaluation of the FPGA implementation of RTM.



(a) Computation only (b) Computation + I/O

Figure 4.5: Elapsed times for different platform implementations of RTM.

4.4.3 FPGA

We also compare our implementation to a FPGA-based implementation for the Convey HC-1 system 4.4. This implementation of RTM is based on a streaming approach in which the volumes are partitioned into subvolumes which are then streamed through the FPGA. Internally, the FPGA sweeps the planes in the Y dimension of the subvolume. Externally, these subvolumes need to be transferred from the host memory through PCIe to the FPGA internal's Block-RAM. The performance in FPGAs is limited by the size of the computation units and, therefore, how many of them can be instantiated. Three compute units can be implemented in each of the four Virtex5 LX330 devices present in the modeled FPGA platform.

4.5 Performance evaluation

We have carried out experiments to verify first the numerical soundness, and second the performance of the implementation. The experimental results show the superior performance of GPUs compared to the Cell B.E and traditional CPUs. GPUs are also able to match the performance of specialized hardware implementation by the means of FPGAs, with much simpler programmability. The results are averages over repeated runs, to eliminate spurious effects (e.g., bus traffic, or unpredictable operating system events). Figure 4.5a shows the execution time of RTM when no I/O is performed. All the accelerators outperform the homogeneous multicore. The performance is up to 6× faster for the Cell B.E., up to 11× for the Convey HC-1 FPGA based systems, and up to 24× for the NVIDIA Tesla C1060. The Tesla

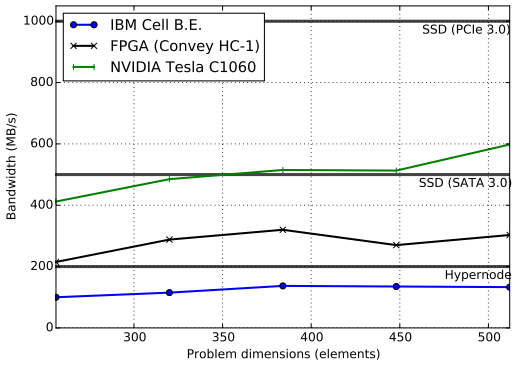


Figure 4.6: I/O requirements of RTM using a stack rate 5, and high level of compression.

C1060 outperforms all other accelerators because of its highly parallel execution model and its higher memory bandwidth that allows for a very efficient implementation of the data-parallel kernels in RTM.
Figure 4.5b, includes disk I/O and describes two types of experiments. Experiment type 1 (i.e., $100, 5$) stands for 100 forward and 100 backward time steps, where the velocity model is a volume from 256 up to 512 grid points per side. Also, in type 1 experiments, the wave field is stored/restored and correlated every five time steps. Experiments of type 2 (i.e., $1000, 10$) have the same setup as type 1 experiments, but the number of time steps is 1,000, and the correlation frequency is 10 time steps. The latter kind of experiments are close to what real industrial executions are, both in terms of model size and time step number.
The I/O technologies attached to the tested architectures become an important bottleneck. This is because the accelerators deliver ready-to-be-stored data at a rate that the I/O is unable to handle. Two main approaches can be taken to minimize this problem: increase the stack rate or apply data compression. Nevertheless, both solutions come at the price of lower results accuracy.
These results depend on the actual I/O technology used in each system. Therefore, we also study the I/O requirements of the different implementations in order not to produce stalls on the program execution. Figure 4.6 depicts the I/O requirements for a test case where the stack rate has been set to 5 steps, compression is in place, and the dimension problem ranges from 256 to 512 cubic points. SSD (i.e., Solid State Disk) is a storage technology that uses flash memories instead of rotating magnetic disks. They are commonly presented as SATA devices, although some SSDs for HPC systems are connected directly to the PCIe interconnect which offers higher bandwidth. Hypernode is a technology proposed by IBM for providing high-performance

I/O for the Cell B.E. platform, which provides a performance similar to a SATA 2 10,000 RPM disk. As can be observed, Hypernode can only handle the data transfer rate for the Cell B.E. processor. For the FPGA-based system an SSD device is required, and the GPU system requires a SSD PCIe device.
Thus, we can say I/O is the main limiting factor for RTM. Furthermore, this problem will become even worse with the utilization of more powerful GPUs, or a higher number of GPUs per node. Current solutions include the utilization of RAID0 configurations that aggregate the bandwidth of several storage devices.

4.6 Summary

We have presented a high-performance implementation of the RTM computation on GPUs. In order to minimize memory transfers between host and GPU memories, all computations (even small serial computations) are performed on the GPU. GPUs outperform all the other accelerators by a large margin, while having a simpler GPU kernel code. I/O becomes the new performance bottleneck, leading to more agressive compression schemes and the utilization of more expensive disk I/O technologies.

Chapter 5

Heterogeneous Multi-GPU Execution

In this chapter we propose HPE, a programming model that simplifies the programmability of multi-GPU systems while improving their performance in many scenarios. We use the multi-GPU parallelization of RTM as a guiding example, although many other applications can benefit from HPE. The research presented in this chapter started in 2009. It has been implemented and tested extensively in several generations of HPE runtime systems as well as adopted into the NVIDIA GPU hardware and drivers for CUDA 4.0 and beyond since 2011. A reduced version of HPE for OpenCL has been also developed in collaboration with Multicoreware Inc. and is shipped in the AMD APP SDK.

5.1 Programmability issues under CUDA

When we started this work, the version 3 of CUDA was available and we use it as a baseline. CUDA 3 presents several restrictions that complicate the development of multi-GPU applications:

- Each thread is bound, for its entire lifetime, to a single GPU. This model forbids direct GPU↔GPU communication, because the CPU thread can only be bound to the source or destination GPU. Therefore, different CPU threads need to collaborate in order to implement this operation, which introduces additional synchronization primitives.
- Each GPU has its own address space. This limitation makes it impossible to determine the location of memory objects solely by inspecting

their address, and programmers need to make sure to which GPU memory each allocation belongs to. Furthermore, GPUs can only access their memory.

- Separate CPU/GPU memory domains. Thus, programmers need to use separate memory allocations in the host and GPU memories and keep them coherent through explicit memory transfers. Communicating GPUs with I/O devices are not supported, either.

Now we show how these limitations impact on the programmability of the multi-GPU implementation of RTM.

5.1.1 RTM on multiple GPUs

The memory requirements of a single shot of the RTM computation can easily exceed the capacity of a single GPU. In order to distribute the execution of a shot, a two-level domain decomposition is applied. First, we decompose the wave field volumes (remember that the current and the two previous time steps are required) into subdomains and distribute them across different nodes. Second, each node's subdomain is further decomposed into smaller subdomains that are handled individually by each GPU in the node. Domain decomposition is performed on the Y dimension of the volumes. Thus, data that needs to be exchanged is contiguous in memory and can be communicated with a single transfer. Otherwise, hundreds or thousands of smaller transfers must be programmed. Besides the costs of the large number of the library calls, PCIe requires large transfers to achieve good efficiency (transfer rates have been shown in Figure 2.4, in page 14).
We use MPI to communicate data between GPUs that are on different nodes. MPI is de facto standard to communicate data in HPC clusters. In MPI, several instances of the program (i.e., processes) are created in the cluster and allow the instances to communicate regardless their physical location. Thus, MPI hides the complexity of the system interconnect topology and inter-node routing. While a program can be written in a way that each MPI process manages a single GPU, this causes suboptimal inter-GPU communication for GPUs that reside in the same node. This is because each process can only access the memory of the GPU bound to it and, therefore, intermediate copies on the host memory would be performed by the runtime system. Therefore, a single MPI process is used per node and each MPI process explicitly manages all GPUs in the node.
Each subdomain has its own halo regions, but some of them refer to the real boundaries of the original wave field, while halo regions in the decomposed

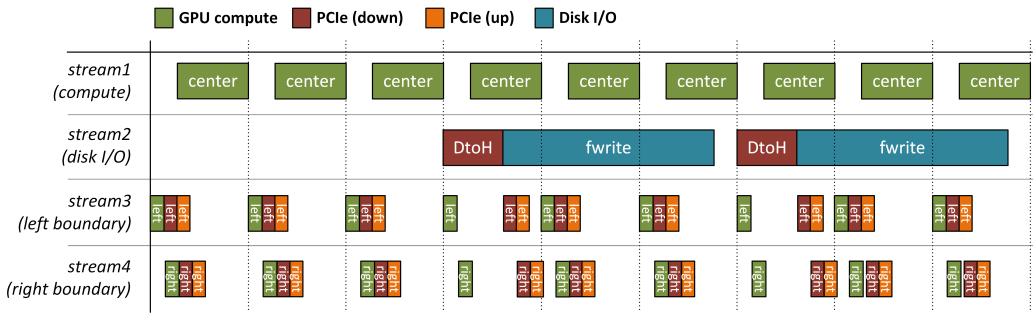


Figure 5.1: Execution timeline for the multi-GPU implementation of RTM.

dimension correspond to elements of the neighboring subdomains. At the end of every simulation time step, data needs to be exchanged between GPUs so that these halo regions contain up-to-date values.

With respect to the domain boundary exchange, the limitation from CUDA 3 makes impossible for a single thread to trigger a GPU-to-GPU memory transfer. Therefore, two threads collaboratively perform the transfers using a two-stage proxy pattern. Two semaphores per neighbor are used: one notifies that the halo region is available in host memory, the second one notifies that the boundary in GPU memory has been updated. In order for these transfers to be performed in parallel to kernel execution, two additional streams per GPU are required. Moreover, as explained in the previous chapter, disk I/O needs to be overlapped with the execution of the next time steps, too. In the multi-GPU version, an additional *I/O thread* and the two semaphores for synchronization are used per GPU. Figure 5.1 shows an example timeline of the used scheme with the streams used by each GPU, using a stack rate of 3. Note that boundary exchange transfers need to wait to the GPU↔host transfers performed by the I/O thread, as they share the down link of the PCIe interconnect.

Listing 5.1 shows the CUDA host code for one simulation step of the RTM loop assuming the CUDA 3 programming model. For simplicity we focus on the 3D stencil computation and the inter-GPU boundary exchange, and the rest of the kernels and the disk I/O scheme are not included there. Therefore, `stream2` does not appear in the code. Figure 5.2 shows a diagram with the steps required to perform the boundary exchange and the synchronization scheme used by the CPU threads in each MPI process. The `id` variable identifies the CPU thread within the MPI process, while `global_id` is the identifier of the CPU thread across all the MPI processes of the application. Before the main loop, the application allocates four semaphore arrays (`l_bound_sem[]`, `r_bound_sem[]`, `l_halo_sem[]`, `r_halo_sem[]`), two arrays of host memory buffers (`l_bound_host[]`, and `r_bound_host[]`), and two extra

```
for (int i = 0; i < time_steps; ++i) {
   launch_stencil_kernel(&out[id][left_off],   &in[id][left_off],   size_left, stream3);
   launch_stencil_kernel(&out[id][right_off],  &in[id][right_off],  size_right, stream4);
   launch_stencil_kernel(&out[id][center_off], &in[id][center_off], size_center, stream1);

   if (global_id > 0) {                     // Left  boundary to host (1a)
      cudaMemcpyAsync(l_bound_host[id], &in[id][l_bound_off],
                      size, cudaMemcpyDeviceToHost, stream3);
      cudaStreamSynchronize(stream3);
   }
   if (global_id < last_id) {               // Right boundary to host (1b)
      cudaMemcpyAsync(r_bound_host[id], &in[id][r_bound_off],
                      size, cudaMemcpyDeviceToHost, stream4);
      cudaStreamSynchronize(stream4);
   }
   if (id > 0)                                 // Right halo is ready (2a)
      sem_post(&r_halo_sem[id-1]);
   if (id < num_gpus - 1)                      // Left  halo is ready (2b)
      sem_post(&l_halo_sem[id+1]);

   if (id == num_gpus - 1 && global_id < last_id) {
                                          // MPI send right and receive left
      MPI_SendRecv(r_bound_host[id], size, MPI_FLOAT,
                   r_neighbor, 0, MPI_COMM_WORLD,
                   l_halo_host, size, MPI_FLOAT,
                   l_neighbor, 0, MPI_COMM_WORLD, &status);
      if (l_neighbor != MPI_NULL)
         cudaMemcpyAsync(&in[id][l_halo_off], l_halo_host,
                         size, cudaMemcpyHostToDevice, stream3);
   }
   if (id == 0 && global_id > 0) {       // MPI send left and receive right
      MPI_SendRecv(l_bound_host[id], size, MPI_FLOAT,
                   l_neighbor, 0, MPI_COMM_WORLD,
                   r_halo_host, size, MPI_FLOAT,
                   r_neighbor, 0, MPI_COMM_WORLD, &status);
      if (r_neighbor != MPI_NULL)
         cudaMemcpyAsync(&in[id][r_halo_off], r_halo_host,
                         size, cudaMemcpyHostToDevice, stream4);
   }

   if (id > 0) {                                    // Update left  halo
      sem_wait(&l_halo_sem[id]);                                  // (3a)
      cudaMemcpyAsync(&in[id][l_halo_off], r_bound_host[id-1],    // (4a)
                      size, cudaMemcpyHostToDevice, stream3);
      cudaStreamSynchronize(stream3);
      sem_post(&r_bound_sem[id-1]);                               // (5a)
   }
   if (id < num_gpus - 1) {                         // Update right halo
      sem_wait(&r_halo_sem[id]);                                  // (3b)
      cudaMemcpyAsync(&in[id][r_halo_off], l_bound_host[id+1],    // (4b)
                      size, cudaMemcpyHostToDevice, stream4);
      cudaStreamSynchronize(stream4);
      sem_post(&l_bound_sem[id+1]);                               // (5b)
   }
   if (id > 0)                               // Wait for left  neighbor (6a)
      sem_wait(&l_bound_sem[id]);
   if (id < num_gpus - 1)                    // Wait for right neighbor (6b)
      sem_wait(&r_bound_sem[id]);
   cudaStreamSynchronize(stream1);

   tmp = in; in = out; out = tmp;                    // Exchange pointers
}
```

Listing 5.1: Simplified host code of the RTM computation using the CUDA 3 programming interface.

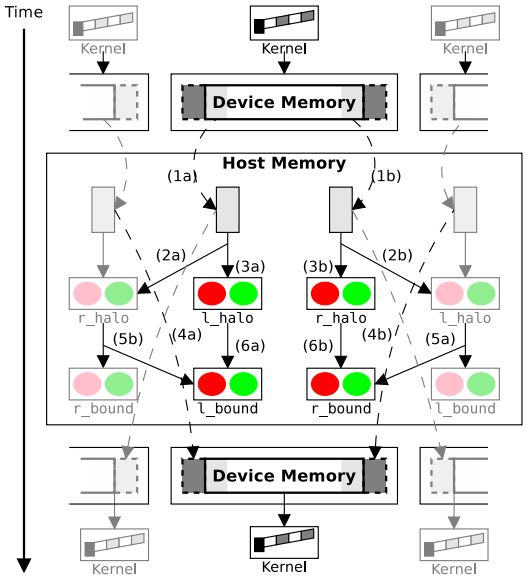


Figure 5.2: Exchange steps and synchronization in the RTM computation using the CUDA 3 programming interface. Legend: solid arrows represent semaphore updates and dashed arrows represent data copies.

buffers for the data received from MPI calls (`l_halo_host` and `r_halo_host`). The size of the semaphore and host memory buffer arrays is equal to the number GPUs.

At the beginning of each time step, we first compute the values in the boundaries so that they can be transferred as soon as possible (lines 2 and 3). Then, the kernel that computes the rest of the wave field is launched (line 4). In the first stage of the exchange, the CPU thread copies the boundaries from the GPU to host memory buffers (`l_bound_host[id]` in line 7 and `r_bound_host[id]` in line 12). These copies are shown as arcs (1a) and (1b) in Figure 5.2. Once the left boundary data has been copied to a host memory buffer, the CPU thread signals its left neighbor that new data is available by posting the `r_halo_sem[id−1]` semaphore (line 17 and arc 2a). An analogous synchronization mechanism (`l_halo_sem[id+1]` semaphore) is required for the right boundary data (line 19 and arc 2b). Outermost threads perform a data exchange using `MPI_SendRecv` (lines 23 and 32) and update their halos with the received data (lines 28 and 37).

The CPU thread starts the second stage by waiting for its neighbor to finish copying boundary data to the host memory buffers (`r_bound_host[id−1]` and `l_bound_host[id+1]`). This is done by waiting on the semaphores for these host memory buffers: `r_halo_sem[id]` and `l_halo_sem[id]` respectively (lines 42 and 49, arcs 3a and 3b). This synchronization is not needed by the outermost boundary data because it is explicitly requested through MPI.

Then, the CPU thread copies the boundary data from each host memory buffer to the halo cells on the GPU (lines 43 and 50, arcs 4a and 4b), signals its neighbors that the data from the host buffer has been consumed (lines 46 and 53, arcs 5a and 5b) and waits, before the next iteration, for the signals from the neighbors (lines 56 and 58, arcs 6a and 6b) using semaphores (`l_bound_sem[id]` and `r_bound_sem[id]`). Finally, `in` and `out` pointers are swapped so the output becomes the input in the next step.

5.1.2 Performance considerations

The boundary exchange in Figure 5.1 assumes that the host buffers for the boundary exchange use pinned memory. In a typical seismic simulation using a 4-point boundary data with a cross section of 2048×2048 single-precision points, 64 MB per boundary are required. However, pinned memory tends to be a scarce resource and so such large host pinned memory requirements can easily harm the system performance. Moreover, data is only updated when the whole boundary has been transferred to host memory by the *owner thread*.

Double-buffering is typically used to improve data transfer rates while reducing the pinned memory requirements. Thus, the region to be transferred is split into smaller subregions, and two pinned buffers are allocated in host memory. It exploits the full-duplex nature of the PCIe interconnect, so that transfers to host memory and transfers to GPU memory overlap. In our implementation, the application allocates two 2 MB host pinned memory buffers per boundary. One of the buffers is used to transfer a block of the source boundary data to the host while the second buffer is used to transfer the previous block to the destination GPU. This implementation mostly hides the cost of data transfers in one of the directions, effectively doubling the data transfer memory bandwidth. In our seismic simulation example, double-buffering reduces the total transfer time of a boundary from 16 ms down to 8 ms.

The performance benefits of double-buffering come at the cost of code complexity. Besides the semaphores in Listing 5.1, two more semaphores are required per host pinned memory buffer. These semaphores protect the contents of the host pinned memory buffer used by ongoing data transfers to the destination GPU, and avoid starting transfers to the destination GPU before the data has been completely transferred to the host memory. It also requires the programmer to insert a new level of loops to iterate over the original exchange code in 2MB transfer steps. The additional synchronization calls, the management of several host memory buffers, and the usage of asynchronous memory transfers greatly increase code complexity.

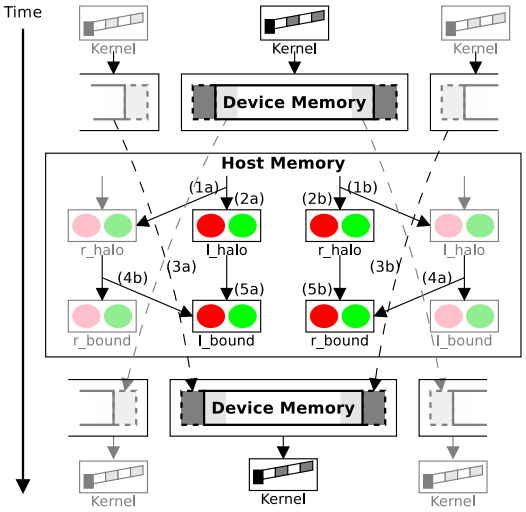


Figure 5.3: Exchange steps and synchronization in the RTM computation when GPUs are shared across CPU threads.

The complexity of the application code increases further if different device system topologies need to be supported. For example, different data exchange code paths are required to avoid memory copies in systems where CPU and GPU share the same physical memory. As a consequence, development and maintenance costs of multi-GPU codes easily become unaffordable. We argue that providing higher-level abstractions is key to exploit heterogeneous GPU-based systems.

5.2 The Heterogeneous Parallel Execution model

In this section we present the Heterogeneous Parallel Execution (HPE) model, built around three novel mechanisms to improve the performance and programmability of heterogeneous multi-GPU systems.

- Multi-threaded GPU sharing.
- Unified virtual address space.
- Multi-GPU and multi-threaded Asymmetric Distributed Shared Memory (ADSM).

Each mechanism is illustrated with CUDA code using the simplified RTM version discussed above. All of the concepts discussed here can be applied to other programming models, heterogeneous devices, and applications.

5.2.1 Multi-threaded GPU sharing

A major limitation of CUDA 3 is the permanent and exclusive binding of only one CUDA context to each CPU thread. Thus, a CPU thread cannot directly copy memory between GPUs (because each GPU hast its own CUDA context). This limitation forces programmers to implement communication between accelerators with intermediate data copies to host memory buffers (i.e., proxy pattern described in Section 5.1.1). That is, the host memory serves as an intermediate switch for routing data from one GPU memory to another GPU memory. Such implementations negatively impacts application performance: a CPU thread requiring data from a GPU bound to another CPU thread must wait until this data is copied to host memory by the other thread. Another undesirable side effect is that both CPU threads might copy their halo data from host memory to GPU memory at almost the same time resulting in PCIe bus contention, giving each CPU thread less than half of the peak PCIe bandwidth.
HPE enables several CPU threads to concurrently access the same GPU context. Listing 5.2 shows the RTM computation code when CPU threads are allowed to access any GPU in the system. Figure 5.3 reflects the updated synchronization scheme used by the CPU threads. The first noticeable modification is that GPU buffers (`in` and `out`) are now global variables accessible by all CPU threads. Now, only the outermost left and right CPU threads must copy the data to be exchanged to host memory (lines 13 and 25, not shown in Figure 5.3 for brevity) since CPU threads within the same node can access the GPU context of its neighbors. Then, they then send this data and receive the updated halo points using MPI (lines 16 and 28).
The local communication code (lines 39 and 47) becomes a GPU-to-GPU memory copy because all CPU threads can access both the source and destination GPU memories. This is enabled by the multi-threaded GPU sharing support in HPE. In this case, each CPU thread waits on the `l_halo_sem[id]` and `r_halo_sem[id]` (lines 38 and 46) for the data produced by other threads to be ready before triggering the data transfer. Once the data exchange is done, each CPU thread notifies the completion of its data copy activity by posting the `l_bound_sem[id-1]` and `r_bound_sem[id+1]` semaphores (lines 43 and 51). Finally, the input and output pointers are exchanged. Notice that this exchange must wait (lines 54 and 56) until after the other threads consume the data because the GPU pointer data structures are shared by several CPU threads.
The first programmability benefit shown in the code in Listing 5.2 is that synchronization points (i.e., calls to semaphores) are conceptually easier to understand. The thread that needs the data performs the `wait` operation

```
for (int i = 0; i < time_steps; ++i) {
   launch_stencil_kernel(&out[id][left_off],   &in[id][left_off],   size_left, stream3);
   launch_stencil_kernel(&out[id][right_off],  &in[id][right_off],  size_right, stream4);
   launch_stencil_kernel(&out[id][center_off], &in[id][center_off], size_center, stream1);

   if (id > 0)                                  // Right halo is ready (1a)
      sem_post(&r_halo_sem[id-1]);
   if (id < num_gpus - 1)                       // Left  halo is ready (1b)
      sem_post(&l_halo_sem[id+1]);

   if (id == num_gpus - 1 && global_id < last_id) {
                                               // MPI send right and receive left
      cudaMemcpyAsync(r_bound_host, &in[id][r_bound_off],
                      size, host, device[id], stream4);
      cudaStreamSynchronize(stream4);
      MPI_SendRecv(r_bound_host, size, MPI_FLOAT,
                   r_neighbor, 0, MPI_COMM_WORLD,
                   l_halo_host, size, MPI_FLOAT,
                   l_neighbor, 0, MPI_COMM_WORLD, &status);
      if (l_neighbor != MPI_NULL)
         cudaMemcpyAsync(&in[id][l_halo_off], l_halo_host,
                         size, device[id], host, stream3);
   }
   if (id == 0 && global_id > 0) {       // MPI send right and receive left
      cudaMemcpyAsync(l_bound_host, &in[id][l_bound_off],
                      size, host, device[id], stream3);
      cudaStreamSynchronize(stream3);
      MPI_SendRecv(l_bound_host, size, MPI_FLOAT,
                   l_neighbor, 0, MPI_COMM_WORLD,
                   r_halo_host, size, MPI_FLOAT,
                   r_neighbor, 0, MPI_COMM_WORLD, &status);
      if (r_neighbor != MPI_NULL)
         cudaMemcpyAsync(&in[id][r_halo_off], r_halo_host,
                         size, device[id], host, stream4);
   }

   if (id > 0) {                                        // Update left halo
      sem_wait(&l_halo_sem[id]);                                  // (2a)
      cudaMemcpyAsync(&in[id][l_halo_off],                        // (3a)
                      &out[id-1][r_bound_off],
                      size, device[id], device[id-1], stream3);
      cudaStreamSynchronize(stream3);
      sem_post(&r_bound_sem[id-1]);                               // (4a)
   }
   if (id < num_gpus - 1) {                            // Update right halo
      sem_wait(&r_halo_sem[id]);                                  // (2b)
      cudaMemcpyAsync(&in[id][r_halo_off],                        // (3b)
                      &out[id+1][l_bound_off],
                      size, device[id], device[id+1], stream4);
      cudaStreamSynchronize(stream4);
      sem_post(&l_bound_sem[id + 1]);                             // (4b)
   }
   if (id > 0)                               // Wait for left  neighbor (5a)
      sem_wait(&l_bound_sem[id]);
   if (id < num_gpus - 1)                    // Wait for right neighbor (5b)
      sem_wait(&r_bound_sem[id]);
   cudaStreamSynchronize(stream1);

   tmp = in; in = out; out = tmp;                      // Exchange pointers
}
```

Listing 5.2: Simplified host code of the RTM computation when GPUs are shared across CPU threads.

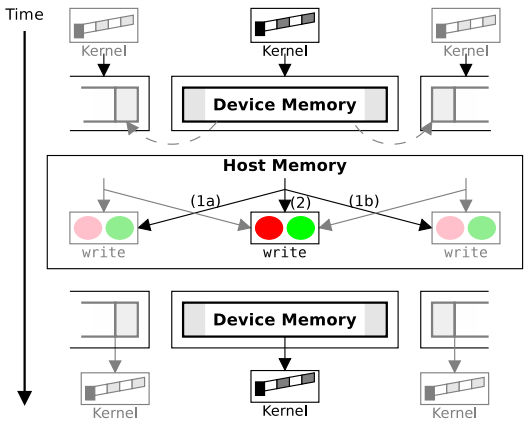


Figure 5.4: Exchange steps and synchronization in the RTM computation when UVAS is available.

just before performing the data exchange, and the `post` operation afterward. This is in contrast to the code in Listing 5.1, where these two semaphore calls are made by separate CPU threads: the proxy thread performs the `wait` call, while the `post` operation is done by the consumer CPU thread.
The second benefit is the simplified abstraction of different system architectures. The application code calls a GPU-to-GPU memory copy. The runtime system provides several hardware-dependent implementations. In a system where compute accelerators share the same physical memory, it is implemented as a single memory copy. However, if both accelerators do not share the same memory and peer-to-peer memory transfers are not supported, the runtime system provides a double-buffered implementation using intermediate host pinned memory buffers. This hardware-independence greatly reduces the amount of code required to achieve high performance in different system architectures. However, programmers still have to explicitly identify source and destination GPUs in the memory copy function.

5.2.2 UVAS and remote memory access

A common characteristic of the codes in Listings 5.1 and 5.2 is that the same virtual memory address might refer to multiple host and GPU memory locations on different computational units (i.e., CPU or accelerator). The most important consequence of virtual memory aliasing is the inability to perform remote memory accesses between GPUs. Neither the host nor the GPUs can determine at runtime the physical memory that a given pointer variable is referring to. Therefore, memory copy operations require a source and destination GPU/host (e.g., `cudaMemcpyHostToDevice` in Listing 5.1, and `device[id]`/`host` in Listing 5.2).

```
1  for (int i = 0; i < time_steps; ++i) {
2     launch_stencil_kernel(out + left_off,   in + left_off,   size_left, stream3);
3     launch_stencil_kernel(out + right_off,  in + right_off,  size_right, stream4);
4     launch_stencil_kernel(out + center_off, in + center_off, size_center, stream1);
5
6     if (id == num_gpus - 1 && global_id < last_id) {
7                                          // MPI send right and receive left
8        cudaMemcpyAsync(r_bound_host, &out[r_bound_off], size, stream4);
9        cudaStreamSynchronize(stream4);
10       MPI_SendRecv(r_bound_host, size, MPI_FLOAT,
11                    r_neighbor, 0, MPI_COMM_WORLD,
12                    l_halo_host, size, MPI_FLOAT,
13                    l_neighbor, 0, MPI_COMM_WORLD, &status);
14       if (l_neighbor != MPI_NULL)
15          cudaMemcpyAsync(&in[l_halo_off], l_halo_host, size, stream3);
16    }
17    if (id == 0 && global_id > 0) {       // MPI send left and receive right
18       cudaMemcpyAsync(l_bound_host, &out[l_bound_off], size, stream3);
19       cudaStreamSynchronize(stream3);
20       MPI_SendRecv(l_bound_host, size, MPI_FLOAT,
21                    l_neighbor, 0, MPI_COMM_WORLD,
22                    r_halo_host, size, MPI_FLOAT,
23                    r_neighbor, 0, MPI_COMM_WORLD, &status);
24       if (r_neighbor != MPI_NULL)
25          cudaMemcpyAsync(&in[r_halo_off], r_halo_host, size, stream4);
26    }
27
28    if (id > 0) {                                // Right halo is ready (1a)
29       cudaStreamSynchronize(stream3);
30       sem_post(&write_sem[id-1]);
31    }
32    if (id < num_gpus - 1) {                     // Left  halo is ready (1b)
33       cudaStreamSynchronize(stream4);
34       sem_post(&write_sem[id+1]);
35    }
36    if (id > 0)                                    // Wait for neighbors (2)
37       sem_wait(&write_sem[id]);
38    if (id < num_gpus - 1)
39       sem_wait(&write_sem[id]);
40    cudaStreamSynchronize(stream1);
41
42    tmp = in; in = out; out = tmp;                      // Exchange pointers
43 }
```

Listing 5.3: Simplified host code of the RTM computation when UVAS is available.

HPE defines a Unified Virtual Address Space (UVAS), where a virtual memory address unequivocally identifies a single location in a GPU/host physical memory. This feature allows the host or any GPU to easily determine the source and destination memories of the memory transfer operations. Coupling the UVAS with hardware support for remote memory transfers, GPUs can transparently access remote memory locations through regular pointers. Listing 5.3 illustrates the programmability benefits provided by the UVAS in our RTM example. Figure 5.4 also shows that the synchronization scheme is much simpler. First, the GPU-to-GPU memory copies that implement the domain boundary exchange between accelerators in the same node are removed, since the kernel code directly accesses the boundary data of the neighboring domains. The kernel launch now receives an additional parameter `id` that identifies the current domain and is used by the kernel code to determine the index of the pointers that belong to the neighboring domains. The outermost CPU threads must still copy the boundary data to an intermediate host memory buffer (lines 8 and 18) before exchanging halo data with neighbour MPI processes for the next iteration (lines 10 and 20). Another host to GPU copy is needed to update the halo data in the corresponding GPU memories (lines 15 and 25). Then, all CPU threads signal that the boundary data (i.e., halo data for other CPU threads) is available for the next kernel call using the `write_sem` semaphore of the left and right neighbors (lines 30 and 34, arcs 1a and 1b). Finally, each CPU thread waits for the neighbor CPU threads to finish (lines 37 and 39) before exchanging the input and output pointers.
Another benefit of the UVAS is that fewer parameters are required by memory copy calls (lines 8, 15, 18 and 25). The UVAS enables the runtime system to determine both the source and destination GPU/host of a memory copy by inspecting the source and destination addresses, eliminating the need for programmers to specify it.

Implementation notes

In order to provide UVAS in systems with pre-Fermi GPUs, we propose a software implementation based on memory segmentation (Figure 5.5). A virtual memory subspace is assigned to the host and each GPU present in the system. The maximum size of each memory subspace is given by the number of bits in GPU physical addresses (e.g., 40 bits for NVIDIA GPUs, 1 TB). These memory subspaces only contain mappings for data hosted in one physical memory. We use the upper bits of the virtual address to identify the GPU where the data is hosted. The HPE runtime assigns a bit pattern to each GPU in initialization time. On API calls taking pointers as input

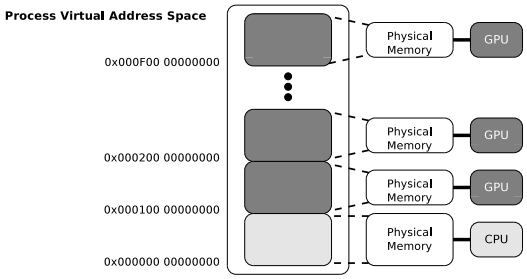


Figure 5.5: Software-based Unified Virtual Address Space implementation using segmentation.

parameters (e.g., memory copy operations), HPE determines the virtual address subspaces involved in the operation. In the GPU code, those bits that identify the virtual address subspace must be discarded in each memory access. Some processors already ignore the upper bits of the address, otherwise this transformation can be transparently inserted by the compiler. For example, a pointer to virtual address `0x000200 00001000` will be truncated to `0x00 00001000`, which is a valid GPU physical address, and `2` will be used to identify the GPU that holds the data.
However, using the software segmentation technique has some limitations. For example, since each virtual address space is mapped to the whole continuous physical address space of an accelerator, it is not possible to transparently distribute data structures by mapping different virtual address ranges across GPUs. Therefore, data structures must be split into chunks and use different pointers to access the appropriate chunks. Conversely, GPUs with virtual memory support (e.g., Fermi and later) can have a continuous representation of a distributed data structure in the UVAS and, therefore, only need a single pointer for the whole data structure.

5.2.3 Multi-threaded ADSM

Despite the benefits provided by the UVAS, the code in Listing 5.3 still presents a major programmability drawback. As discussed in Section 5.1.1, overlapping MPI transfers with GPU↔host transfers to reduce the communication time is key to achieving high performance. This optimization requires different source code paths for systems with separate GPU and host memories, and systems where a single memory is shared among the accelerators and the host.
HPE supports an extension of the Asymmetric Distributed Shared Memory (ADSM) model [45] to multi-threaded and multi-accelerator systems. In HPE, each thread can select its *current* GPU. Memory allocation and kernel

```
1  for (int i = 0; i < time_steps; ++i) {
2     launch_stencil_kernel(out + left_off,  in + left_off,   size_left);
3     launch_stencil_kernel(out + right_off, in + right_off,  size_right);
4
5     if (id > 0)                                  // Right halo is ready (1a)
6        sem_post(&write_sem[id-1]);
7     if (id < num_gpus - 1)                       // Left  halo is ready (1b)
8        sem_post(&write_sem[id+1]);
9
10    if (id == num_gpus - 1 && global_id < last_id) {
11          // MPI send asynchronously right and receive asynchronously left
12       MPI_Isend(&out[r_bound_off], size, MPI_FLOAT,
13                 r_neighbor, 0, MPI_COMM_WORLD, &req[0]);
14       MPI_Irecv(&in[l_halo_off], size, MPI_FLOAT,
15                 l_neighbor, 1, MPI_COMM_WORLD, &req[1]);
16    }
17    if (id == 0 && global_id > 0) {
18          // MPI send asynchronously left and receive asynchronously right
19       MPI_Isend(&out[l_bound_off], size, MPI_FLOAT,
20                 l_neighbor, 1, MPI_COMM_WORLD, &req[2]);
21       MPI_Irecv(&in[r_halo_off], size, MPI_FLOAT,
22                 r_neighbor, 0, MPI_COMM_WORLD, &req[3]);
23    }
24
25    launch_stencil_kernel(out + center_off, in + center_off, size_center);
26
27    if (id == num_gpus - 1 && global_id < last_id) // Wait for MPI transfers
28       MPI_Wait(req[1], NULL);
29    if (id == 0 && global_id > 0)
30       MPI_Wait(req[3], NULL);
31    if (id > 0)                                    // Wait for neighbors (2)
32       sem_wait(&write_sem[id]);
33    if (id < num_gpus - 1)
34       sem_wait(&write_sem[id]);
35
36    tmp = in; in = out; out = tmp;                      // Exchange pointers
37 }
```

Listing 5.4: Simplified host code of the RTM computation when ADSM is available.

invocations are performed on the *current* GPU of the thread that requests the operation. Each GPU is assigned several internal streams to implement efficient data transfers and computation/memory transfer overlap. In ADSM, operations are synchronous and, therefore, several CPU threads are required to fully utilize more than one GPU. It also eliminates the need for CUDA streams. We define a consistency model for concurrent accesses from different CPU threads, that relies on the common inter-thread synchronization mechanisms. Thus, if one CPU thread needs to access the output generated by a kernel launched by a different CPU thread, synchronization primitives such as semaphores are required. On the other hand, accesses during kernel execution to shared objects that are only read are also allowed. This lets the disk I/O thread access the wave field to be stored while the new wave field is being computed. Accesses during kernel execution to shared objects that are written are undefined. The programmer is responsible for ensuring that these constraints are honored. While memory protection could be used to enforce the constraints, it would introduce some overhead due to the required TLB shootdowns [95]. Moreover, CUDA does not provide any interface to change the protection of the pages in the GPU memory.

Moreover, on systems where accelerators and CPU have separate memories, the runtime system captures MPI calls (e.g., using library interposition) and based on the virtual address passed as source and destination buffers determines the accelerator hosting the data, and double buffers the MPI transfer. This is performed by splitting the MPI transfer into several smaller MPI transfers to overlap them with the GPU↔host transfers. On system architectures where the CPU and compute accelerators share the same physical memory the runtime performs a simple MPI transfer.

Listing 5.4 shows the RTM computation code when the ADSM model is incorporated. The explicit `cudaMemcpyAsync` API calls before and after MPI transfers have been removed. On the other hand, since kernel calls are synchronous in ADSM, non-blocking MPI calls (lines 12, 14, 19 and 21) are required so that they can overlap with the execution of the main kernel (line 25). Later, each CPU thread explicitly waits for the MPI calls (lines 28 and 30) to complete. Finally, the CPU threads waits for the neighbors to consume the boundaries (lines 32 and 34) before proceeding to the next time step.

The ADSM included in HPE raises the level of abstraction; by providing a high-level abstract machine model, the runtime can efficiently map data exchange and I/O operations to all potential heterogeneous system architectures. The simplified MPI calls in Listing 5.4 are an example of the benefits of this higher level of abstraction.

5.3 Performance evaluation of the HPE model

In this section we measure the performance effects of the different features in HPE. The availability of real hardware that support key HPE features gives rise to a rare opportunity for studying the effectiveness of the hardware support by running important benchmarks on real runtime and hardware.

5.3.1 Experimental methodology

All experiments were run on a system containing a dual Intel Intel(R) Xeon(TM) E5620 at 2.40 GHz with 24 GB of DDR3 RAM memory, and 4 NVIDIA C2070 6 GB GDDR5 GPU cards. Mellanox Technologies MT26428 QDR 11 40 Gbps Infiniband network adapters are used in those tests that require network communication. The CPU sockets are connected to different PCIe 2.0 32x buses, each connected to two GPUs. Peer-DMA is enabled for GPUs on same bus. All machines run a GNU/Linux system, with Linux kernel 3.8.0 and NVIDIA driver 304.88. Benchmarks were compiled using GCC 4.7.3 for CPU code and NVIDIA CUDA compiler 5.5 for GPU code. Execution times were measured using `gettimeofday`, which offers a μsecond granularity, for the host code and CUDA events for GPU code and memory transfers. All results show the average of 30 runs; samples higher than the arithmetic average plus/minus the variance were considered outliers and discarded.
We evaluate the performance impact of the HPE features using the CUDA runtime and GMAC. GMAC provides multi-threaded ADSM, not implemented in CUDA 5.5, which is the version available at the time of this evaluation.

5.3.2 Benchmarks

Two synthetic benchmarks were used to characterize inter-GPU data copies. The first benchmark is a one-way data copy from a source GPU to a destination GPU. This communication pattern is found in n-body simulations, where the particles moving out from one domain are sent to the neighbouring domain. This benchmark produces no contention on the PCIe bus, which provides an environment where software locking costs can be measured. The second benchmark is a two-way data copy between GPUs. This inter-GPU communication pattern is found in most multi-GPU computations. Applications present a wide range of data exchange sizes; for instance, waveguide simulation typically requires exchanging hundreds of kilobytes, fluid dynamics simulations tens of megabytes, and FFTs hundreds of megabytes. To

account for these scenarios, experiments were run using data exchange sizes ranging from 256 KB to 256 MB. These benchmarks also evaluated the locking overhead required for multi-threaded GPU sharing. Experiments are run for different communication schemes (naive, pinned, double-buffered) implemented on CUDA with no HPE features (CUDA-base), and GMAC (HPE) with and without peer-DMA support.
The performance of HPE was also measured using real-world applications. Current benchmark suites for GPUs like Parboil [48] and Rodinia [35] target single-GPU systems and do not stress data communication. Therefore, we have developed CUDA (using the previously mentioned communication schemes) and HPE versions of the following well-kown computations. We use a CPU thread for each GPU in the node. Each of these threads launches kernels on their assigned GPU but also access other GPUs to perform GPU↔GPU memory transfers when needed. **fdtd** is the part of the RTM application introduced in Section 5.1.1 that uses 3D finite differences. 1D FFT application (**fft**) implements a Fast Fourier Transform on a 1D input vector using the Radix-2 Cooley-Tukey algorithm. This algorithm performs n steps in which different elements are combined. The combination pattern changes at each step and, therefore, in the multi-GPU implementation, data must be exchanged between different pairs of GPUs at each step. We use the multi-GPU implementation of **mergesort** found in [89]. The input vector is divided into chunks that are individually sorted by each GPU. Then, a *swap* phase merges the subvectors into a sorted vector whose contents are logically distributed among the memories of the GPUs.
We have also developed two synthetic benchmarks to convey the benefits of the techniques implemented in our HPE runtime to optimize the communication with I/O devices. The first benchmark measures the time needed to transfer a file from disk to the GPU memory using four different implementations: *user* uses a regular user-level allocation to store the contents of the file and then transfer it to the GPU memory; *pinned* uses pinned memory instead of a user-level allocation; *double-buffering* uses two small pinned buffers to minimize the usage of pinned memory and to overlap the disk and GPU memory transfers. The second benchmark measures the time needed to send data across GPUs in different nodes through MPI. The following configurations are compared: *user* uses a regular user-level allocation to store the contents of the transfer before calling to send/receive data from the network; *pinned* uses pinned memory instead (it exploits the GPUDirect technology that enables Infiniband interfaces to use the pinned memory allocated through CUDA); *HPE* uses two small pinned buffers to overlap network and CPU↔GPU transfers.

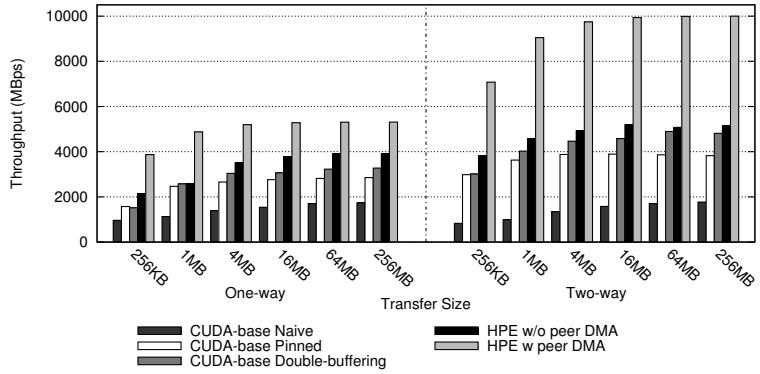


Figure 5.6: Measured throughput for different data communication sizes.

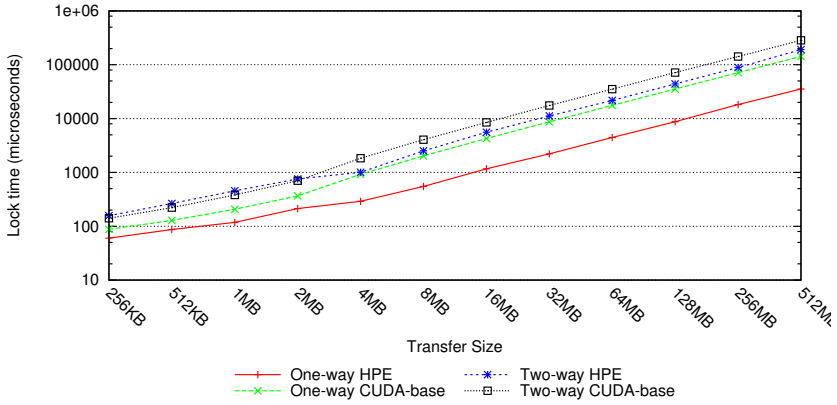


Figure 5.7: CPU thread wait time for different inter-GPU data communication sizes.

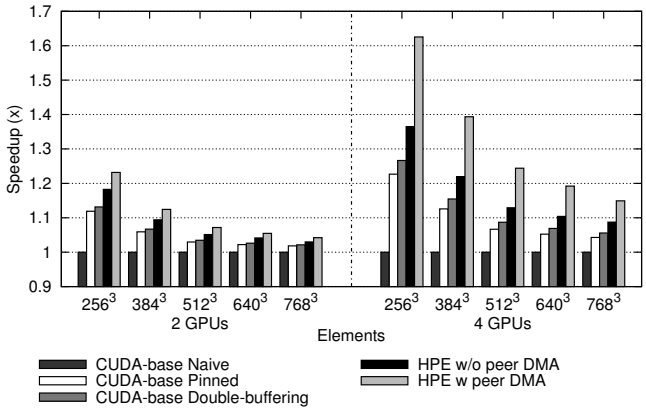
5.3.3 Inter-GPU data transfers

Figure 5.6 (left) shows the one-way inter-GPU communication throughput delivered by each implementation for different communication sizes. Peer-DMA always delivers the highest throughput because there are no associated software communication overheads. Peer-DMA also delivers the highest throughput for two-way data exchange, as shown in Figure 5.6 (right). For a one-way communication, HPE with no peer-DMA support delivers 70% compared to hardware peer-DMA HPE for large data communication sizes due to the costs of performing intermediate copies to the host memory. However, Figure 5.6 (right) shows that the throughput delivered by hardware peer-DMA is almost 2× faster than the software emulated peer-DMA for a two-way data exchange. This additional performance penalty is due to contention for exclusive access to the source and destination GPU contexts, which are being used concurrently by all CPU threads.

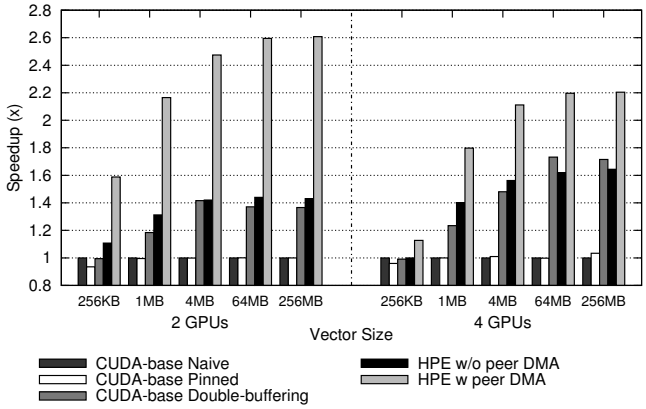
Figure 5.6 (left and right) also shows that the double-buffering strategy is always the optimal software implementation. The benefit of double-buffering becomes noticeable for data communications larger than 1 MB, when double-buffering starts overlapping of host-to-GPU and GPU-to-host data transfers. The *pinned* implementation transfers all the data to a pinned buffer in host memory and, therefore, does not overlap data transfers. Still, the throughput delivered by this scheme is a 40% higher than the base implementation.
Figure 5.7 shows the total wait time for HPE and CUDA-base double-buffering. Locking time in HPE is shorter than in CUDA-base double-buffering, except for two-way communication of small halo sizes. HPE only requires exclusive access to the GPU context for the duration of the API call that enqueues an asynchronous data transfer between the host and the GPU; after the DMA command has been requested to the hardware, the PCIe configuration registers can be used to request new DMA transactions. This is in contrast with the double-buffering implementation in CUDA-base, which requires exclusive access to the intermediate host buffer for the duration of each data transfer. For small size data transfers, the total time CUDA-base requires exclusive access to the intermediate buffer is short (few data is transferred) and it shows a better behaviour than HPE.
As the data communication size increases, CUDA-base requires locking the intermediate buffers for longer times, so the time each CPU thread waits to initiate the next data transfer increases. Inter-GPU data communication in HPE does not require waiting for any other CPU thread to bring the data to the intermediate host buffers. Hence, the lock time due to exclusive access to the GPU context only grows on the number of API calls required for the communication, which grows linearly with the data communication size. The smaller locking time for medium and large inter-GPU communication accounts for the extra throughput provided by HPE.

5.3.4 Application benchmarks

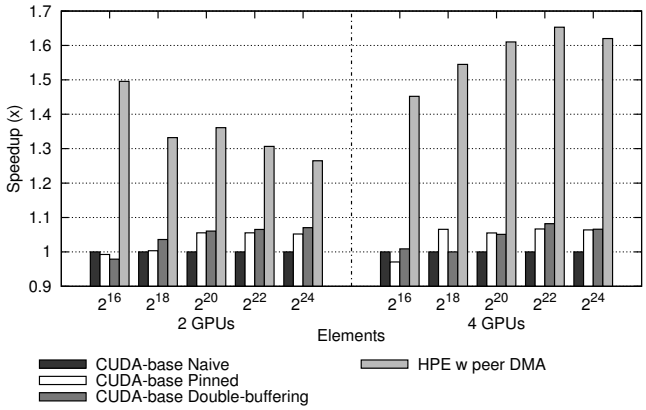
Figure 5.8a shows the speedup of different implementations of the FDTD computation over the base CUDA-base version. In this application, boundaries are exchanged in every iteration with the *left* and *right* neighbors. Due to the limited support of remote accesses in current NVIDIA GPUs (only works for GPUs in the same PCIe bus), we use explicit data transfers in all the implementations. The elements to be exchanged are computed first and are transferred concurrently with the rest of the computation, in order to hide the data transfer costs. The size of the data being transferred goes from 1 MB to 9 MB for the tested input datasets. Results show speedups that range from 1.04× to 1.23× for 2 GPUs and from 1.15× to 1.63× for 4 GPUs



(a) FDTD



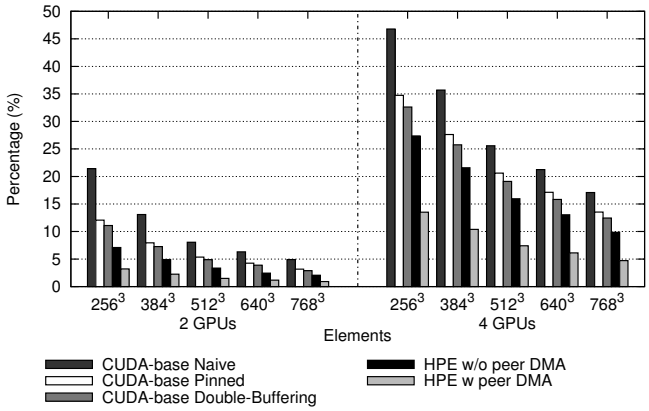
(b) 1D FFT



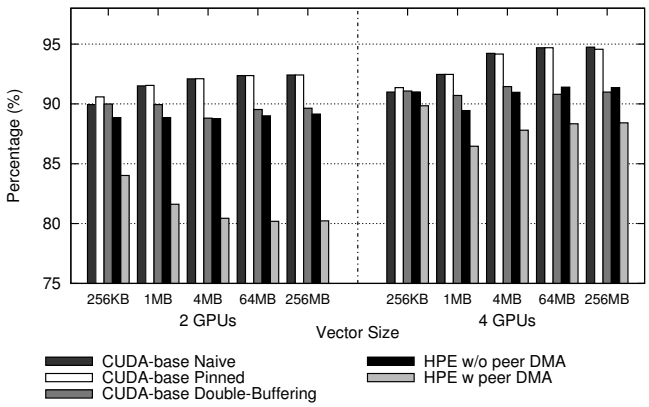
(c) Mergesort

Figure 5.8: Speedup over single GPU execution different input dataset sizes.

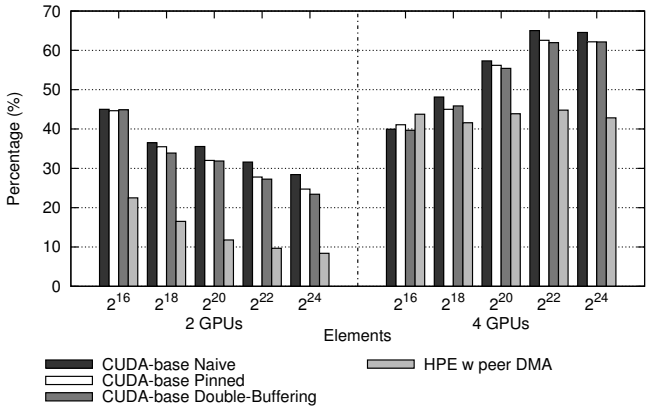
in the HPE version when peer-DMA is available. The largest improvements are obtained for small to medium input datasets, where the data transfer/ computation ratio is high, as shown in Figure 5.9a. In these cases, the data transfer cannot be completely masked. The improvement is greater in the 4-GPU configuration because, in the general case, each domain exchanges twice as data as the 2-GPU case and, therefore, the improvements in the memory transfers are more pronounced.
The 1D FFT application is communication bound and, therefore, greatly benefits from the HPE model. As shown in Figure 5.9b, all configurations except HPE with peer-DMA support spend at least 86 % of the application time in data exchange routines, when using 2 GPUs and 87% for 4 GPUs. On hardware with peer-DMA, HPE exchange time is reduced to 80% for 2 GPUs and is at least 5% lower than the base implementation for 4 GPUs. This reduction results in speedups (see Figure 5.8b) that range from 1.58× to 2.6× for 2 GPUs over the naive CUDA 3 implementation and 1.17× to 1.61× for 4 GPUs, for vector sizes from 256 KB to 256 MB. HPE with no peer-DMA support delivers speedups of 1.15× to 1.42× performance improvements over the base version for 2 GPUs and 1.01× to 1.6× for 4 GPUs (except for the smallest input dataset, since a single buffer is transferred). The performance of the double buffering implementation and HPE for 4 GPUs is closer because peer-DMA transfers across different PCIe buses are not currently supported, so intermediate copies are performed on the host memory. Moreover, the effect of the peer-DMA transfers is limited when using 4 GPUs because our FFT algorithm performs memory swaps between pairs of GPUs and these may be serialized if they involve the same GPU.
Mergesort shows notable speedups when using HPE with peer-DMA transfers. This application swaps the locally sorted subarrays between GPUs in order to be merged into the final sorted array. When 4 GPUs are used, a first swap step is performed between GPUs 0 and 1 and GPUs 2 and 3 (to produce two sorted subarrays) and a final swap step is performed to merge them into the final array. Figure 5.8c reports speedups that range from 1.25× to 1.50× for 2 GPUs and 1.45× to 1.66× for 4 GPUs. This benchmark also benefits from remote accelerator memory access, but current hardware restricts the utilization of this mechanism to GPUs connected to the same PCIe bus. Remote memory accesses are used during the pivot search to determine which data needs to be exchanged between pairs of GPUs. Using remote memory accesses delivers much better performance than copying the necessary data across GPUs (required by HPE when GPUs connected to different PCIe buses, and by CUDA 3). Figure 5.9c shows that the percentage of time devoted to communication decreases as the dataset increases for 2 GPUs. The additional communication steps required in the 4 GPU implementation make



(a) FDTD



(b) 1D FFT



(c) Mergesort

Figure 5.9: Percentage of time devoted to memory transfers over the total execution time.

the communication/computation ratio increase with the input dataset size.

5.3.5 Communication with I/O devices

The use of pinned memory is key for achieving fast transfers between I/O devices and the host memory. Pinned memory becomes even more important for transfers between devices (e.g., disk and GPU). Since these devices usually have their own private address spaces, they rely on intermediate copies to host memory for communication. If user-level memory allocations are used, the Operating System has to perform a number copies to/from these allocations to (system-managed) pinned buffers before starting a DMA transfer. Figure 5.10 shows that pinned memory is 2.2× to 2.7× faster than the base version. Overlapping some of the disk and GPU transfers provides even better performance (2.5× to 3.4×). The HPE runtime matches the hand-tuned implementation (2.4× to 2.9×) while hiding the complexity of this technique.

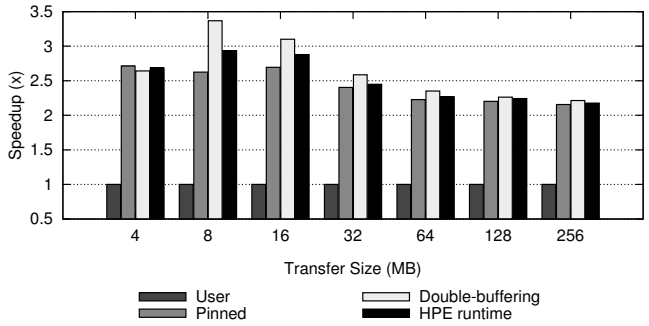


Figure 5.10: Disk↔GPU transfer speedups of HPE compared to the base synchronous version.

5.3.6 Inter-node communication (MPI)

Communicating GPUs across nodes requires moving data between the GPUs' memories and the buffers in the network interfaces. While future systems will be able to perform P2P (i.e., peer-to-peer) transfers between them, currently data has to be stored in host memory. Moreover, pinned memory must be used to avoid extra copies in the network interface driver. Figure 5.11 shows that using pinned memory provides up to 2× better performance in GPU→GPU and GPU→host memory transfers, and up to 2.6× in host→GPU, than regular pageable allocations. Furthermore, the double-buffering implemented in the HPE runtime for some MPI calls further improve the performance, providing speedups greater than 4×.

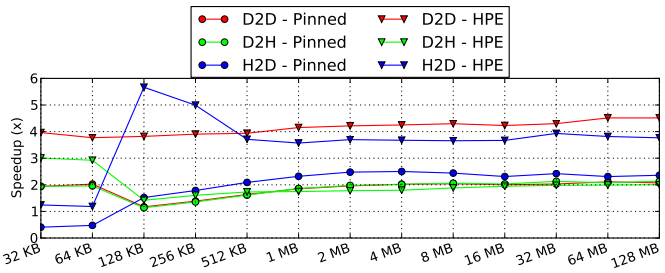


Figure 5.11: MPI transfer speedups of HPE compared to the base synchronous version.

5.4 Summary

HPE greatly simplifies the task of programming multi-accelerator applications for heterogeneous parallel systems, by removing the need for applications to explicitly perform intermediate copies and complex synchronization patterns. We have implemented the HPE model and its associated techniques into GMAC, a user-level library that is publicly available at [10]. Since CUDA 4.0, NVIDIA started to introduce most of the proposed features presented here. AMD also ships a version of the HPE model for OpenCL in its APP SDK. This provided us with an opportunity to quantify the benefit of these features on real hardware. Experiments show that the HPE runtime transparently exploits GPUs that comes with hardware support for HPE features and significantly improves performance when such hardware is present in the system. We show that simple, portable application code based on HPE often achieves performance comparable to complex custom-written code even in systems that do not have good hardware support for HPE techniques. We have outlined the GPU hardware support required to efficiently implement the proposed UVAS. We further argue that without simple interfaces like HPE, the advanced GPU hardware support will unlikely be used by most software applications in practice.

Experimental results show that the HPE model eases programming of multi-accelerator applications while providing performance improvements of 2× compared to the best data transfer scheme implemented on top of CUDA 3. We have also analyzed the impact of HPE on three real benchmarks. Improvements range from 5% in compute-bound benchmarks and up to 2.6× in communication-bound benchmarks. Finally, experiments show that HPE transparently implements sophisticated communication schemes that can deliver up to a 2.9× speedup in I/O device transfers.

5.5 Impact on CUDA 4/5/6

CUDA 4.0 introduced the UVAS support for Fermi and later GPUs. Thanks to the UVAS, the runtime can determine the location of GPU allocations and a single thread can initiate transfers between different GPUs. Moreover, programmers do not need to specify the direction of the transfers in `cudaMemcpy` (`cudaMemcpyDefault` is used by default). Memory allocations can be made visible to other GPUs by synchronizing the GPU memory page tables (using `cudaDeviceEnablePeerAccess`). Hardware support since Fermi (advertised as GPU Direct [5]) transparently routes requests to data located on a different GPU memory through the PCIe interconnect.
CUDA 4.0 also implements the floating context model for host threads. Thus, threads can use the `cudaSetDevice` function to dynamically select its current GPU for implicit operations such as memory allocation and kernel memory launches. For explicit operations that take streams or events, the runtime internally switches to the proper GPU context. Programs are no longer forced to use several threads in order to exploit more than one GPU.
Moreover, CUDA 6.0 introduces an implementation of the multi-threaded ADSM memory management called UVM. Buffers allocated with `cudaMallocManaged` can be used both in CPU and GPU code. UVM integrates with CUDA streams, unlike HPE, which tries to avoid the utilization of CUDA-specific abstractions. By default, all shared buffers are acquired/ released at kernel call boundaries. However, users can assign memory ranges to streams using `cudaStreamAttachMemAsync`, so that only the memory ranges that are attached to a stream are taken into account when launching a kernel on that stream.

Chapter 6

Shared Memory GPU Programming

Discrete GPUs in modern multi-GPU systems can transparently access each other's memories through the PCIe interconnect. Future systems will improve this capability by including better GPU interconnects such as NVLink. However, remote memory access across GPUs has gone largely unnoticed among application developers, and multi-GPU systems are still programmed like distributed systems in which each GPU only accesses its own memory. This increases the complexity of the host code as programmers need to explicitly communicate data across GPU memories. In this chapter we present GPU-SM, a set of guidelines to program multi-GPU systems like NUMA shared memory systems with minimal performance overheads. This work includes the first exhaustive performance study of remote memory accesses over PCIe, and an analysis of their viability as a mechanism to implement the shared memory model in multi-GPU systems.

6.1 Multi-GPU NUMA systems

Shared memory machines are considered to be easier to program than distributed memory machines. In these machines, all processors can access any data regardless its location and, therefore, programmers are relieved from the obligation of distributing data among *nodes*. They can be Symmetric Multi-Processing (SMP), such as the Sun Enterprise series or Intel Pentium Pro powered machines [37], so that the cost of access is the same for all the memories. Nevertheless, the dominant approach in current systems is Non-Uniform Memory Access (NUMA) machines, in which memories are located

at different distances from the processors. Thus, each processor is typically connected to a *local* memory and to several *remote* memories, that are accessed through an interonnection network. Some early examples of cache coherent NUMA (CC-NUMA) machines are the MIT Alewife [18], Stanford DASH [64], but most modern multi-socket machines are built with commodity processors such as Intel CPUs [69].
In order to overcome the costs of remote accesses, early shared memory machines implemented various latency tolerance mechanisms, such as data prefetching or multithreading [60], in order to hide the extra cost of accessing remote memory. For example, the APRIL processor [19] from the Alewife machine switches between threads on each remote memory request or failed synchronization attempt, similarly to GPUs. Nevertheless, current shared memory processors mostly rely on caching to avoid accesses to remote memories, and software-based solutions are used to further reduce their number [72, 71, 38].
Thanks to the UVAS and P2P technologies, multi-GPU machines can be also programmed like shared memory machines: a kernel launched on a GPU can access the memories of all the GPUs in the system. However, they present some key differences with respect to traditional shared memory machines:

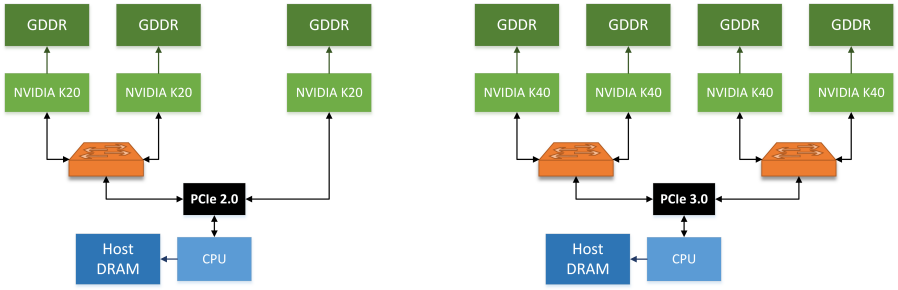
- The costs of remote accesses are much higher on GPUs because of the longer latencies introduced by the PCIe interconnect. Futures interconnects such as NVLink will reduce this ratio, but it will be still greater than in general purpose-based systems.

- Shared memory nodes implement cache coherence between processors to simplify data sharing. GPUs do not implement cache coherence and, therefore, they form a Non Cache Coherent NUMA (NCC-NUMA) system. Thus, only accesses to read-only regions can be cached in GPUs to avoid coherence problems.

- Programming models for GPUs like CUDA provide a weak consistency model between thread blocks. Only atomic operations and memory fences are visible across thread blocks.

- GPUs can handle up to 64 concurrent in-flight warps in each GPU core, with zero context-switch overhead. Thanks to this execution model, GPUs can hide the costs of long latency operations better than general purpose processors.

In this work we try to determine how the unique characteristics of GPUs can help hiding the high costs of remote accesses.



(a) System A (b) System B

Figure 6.1: Multi-GPU systems used to evaluate GPU-SM.

6.2 Experimental methodology

We first analyze the performance characteristics of the remote memory access mechanism on the PCIe interconnect. The perfomance impact of different GPU harwdare features is also measured.

6.2.1 Hardware setup

We use two systems with different PCIe revisions and device topologies.

- *System A* (Figure 6.1a) contains a quad-core Intel i7-930 at 2.8 GHz with 16 GB of DDR3 memory, and 3 NVIDIA Tesla K20 GPU cards with 6 GB of GDDR5 each, connected through PCIe 2.0 in x16 mode. The PCIe topology is 2+1, with two GPUs connected to the same PCIe switch, and the third one to a second switch.
- *System B* (Figure 6.1b) contains a quad-core Intel i7-3820 at 3.6 GHz with 64 GB of DDR3 memory, and 4 NVIDIA Tesla K40 GPU cards with 12 GB of GDDR5 each, connected through PCIe 3.0 in x16 mode.

Both machines run a GNU/Linux system, with Linux kernel 3.16 and NVIDIA driver 340.24. Benchmarks were compiled using GCC 4.8.3 and NVIDIA CUDA compiler 6.5 for GPU code. Execution times are measured using the CUPTI profiling library that provides support for sampling and nanosecond timing resolution.

6.2.2 Microbenchmarks

In order to characterize the hardware platform, we developed a set of microbenchmarks with different computation and memory access patterns.

These microbenchmarks are implemented using a single parametrized GPU kernel that reads from memory, performs several floating point operations on the input data and writes the result to memory. We pass local and remote allocations for both input and output data to each kernel launch and, given some parameters, the code selects the local or the remote pointer. The available parameters are:

- Computational intensity: amount of floating point instructions performed on each input datum (FLOPs/datum). This is implemented using an unrolled loop of dependent operations.
- Amount of remote accesses: percentage of remote accesses relative to the total amount of memory accesses. Threads use their identifiers to determine if they need to access data locally or remotely.
- Remote access pattern: whether accesses are performed by a small set of thread blocks that are scheduled for execution together (*batch*), or they are spread among a bigger set of thread blocks that are executed along with thread blocks that do not perform remote accesses (*spread*).
- Topology: whether remote accesses are performed between GPUs connected to the same or different PCIe switches, or to host memory.
- Occupancy: percentage of warps that execute concurrently on each SM relative to the maximum (i.e., 64). Theoretically, the bigger the occupancy, the more remote accesses can be overlapped with the execution of other threads. We control the occupancy by setting artificial scratchpad memory requirements in the kernel.
- Caching: remote accesses are not cached in the local cache hierarchy. However, remote accesses to read-only data structures can be cached in the private L1 R/O cache in the SM by using the `__ldg` CUDA intrinsic.

6.2.3 Multi-GPU applications

We use two applications in order to test the performance of a GPU-based shared memory implementation: FDTD and Image Filtering. The original and modified implementations are discussed in detail in Section 6.5.

FDTD

This implementation of the finite difference method is a simplified version of the RTM computation introduced in Chapter 4. Domain decomposition

Halo	FLOPs	Occupancy	Volume Size	Grid size	% Remote
2	18	75%	128^3	(4, 32)	6.25%
			512^3	(16, 128)	0.78%
			768^3	(64, 512)	0.52%
4	36	62.5%	128^3	(4, 32)	12.5%
			512^3	(16, 128)	1.56%
			768^3	(64, 512)	1.04%
8	72	56.2%	128^3	(4, 32)	25%
			512^3	(16, 128)	3.12%
			768^3	(64, 512)	2.08%
16	144	31.2%	128^3	(4, 32)	50%
			512^3	(16, 128)	6.25%
			768^3	(64, 512)	4.17%

Table 6.1: Analyzed configurations for the GPU-SM implementation of FDTD.

assigns a portion of the input and output data (i.e., domain) to each GPU in the system.

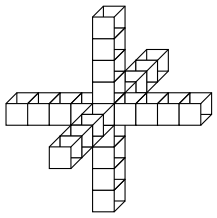


Figure 6.2: Data dependences in a 4-point 3D stencil computation.

The main loop of the finite difference method iterates over the steps of the simulation. For each step, the value of each point in the output volume is calculated by a 3D stencil computation, illustrated in Figure 6.2, that takes as input the value at the point and its neighbors. The output volume of the current step becomes the input for the next step. The stencil computation for the points at the boundaries of a partition (i.e., boundary data) requires input values from neighboring partitions (i.e., halo data). Instead of copying these halo regions, they accessed remotely, as needed.
We run different configurations of the FDTD computation, that are summarized in Table 6.1. Different volume size and halo sizes are tested. The amount of remote memory accesses depends on the volume and halo sizes and range from 0.52% to 50%. Using larger halo sizes increases the number of remote memory accesses, but it also increases the amount of floating point operations (FLOPs) performed per output point. In order to analyze different halo access patterns, we analyze computation decompositions on dimensions X, Y and Z.

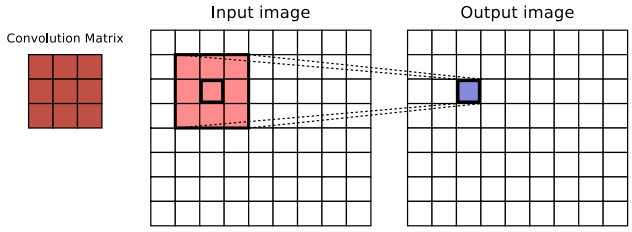


Figure 6.3: Data needed to compute an output pixel in a convolution.

Filter	FLOPs	Occupancy	Image Size	Grid size	% Remote
3×3	18	100%	128^2	(4, 32)	81.27%
			4096^2	(128, 1024)	80.91%
			24576^2	(768, 3072)	80.90%
5×5	36	75%	128^2	(4, 32)	89.58%
			4096^2	(128, 1024)	89.30%
			24576^2	(768, 3072)	89.29%

Table 6.2: Analyzed configurations for the GPU-SM implementation of image filtering.

Image filtering

Image filtering allows to apply different effects that emphasize or remove features from images. It is typically performed in the spatial domain by using a convolution computation, or in the frequency domain by using fast Fourier transform (i.e., FFT). We use the convolution as it is the most efficient method on GPUS [43]. The convolution combines a small convolution matrix (e.g., 3×3 or 5×5) with each input pixel and its neighbors, by multiplying the value of the pixel and its corresponding element of the convolution matrix (Figure 6.3). The value of the output pixel is the sum of all the individual products. Domain decomposition assigns a portion of the image to each GPU. Like in the stencil kernel, the computation pattern creates a halo of data that is required to compute the output value for the pixels in the boundaries. Moreover, the convolution matrix is completely accessed by each GPU in the system. The data sets and filter sizes used in the evaluation are summarized in Table 6.2.

6.3 Performance analysis of the remote access mechanism

In this section we study the characteristics of remote access and how they can impact on the performance of applications. We formulate a number of hypotheses and we test them using the experiments explained in the previous section.

Hypothesis 1 *Remote accesses in GPUs can achieve full PCIe bandwidth. Since PCIe is full-duplex, it can sustain the bandwidth for R/W concurrent remote accesses or accesses of the same type from two different GPUs.*

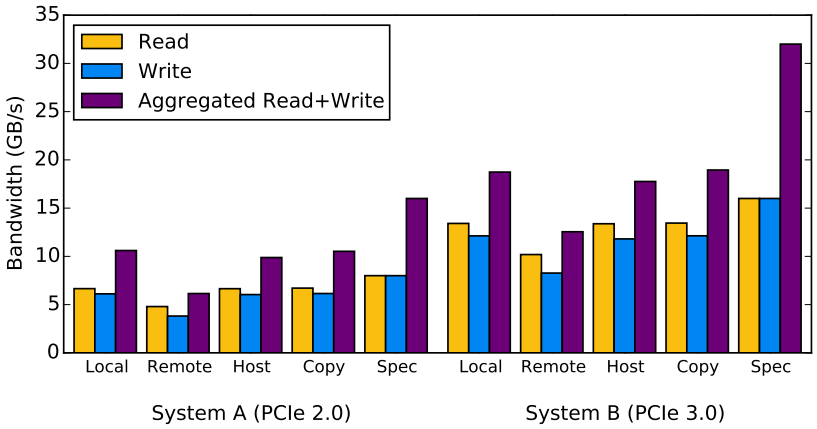


Figure 6.4: Memory bandwidth achieved by remote accesses for different PCIe generations. *spec* is the theoretical peak.

We use a microbenchmark that only performs remote memory accesses to obtain the maximum effective bandwidth. Two different variables are explored:

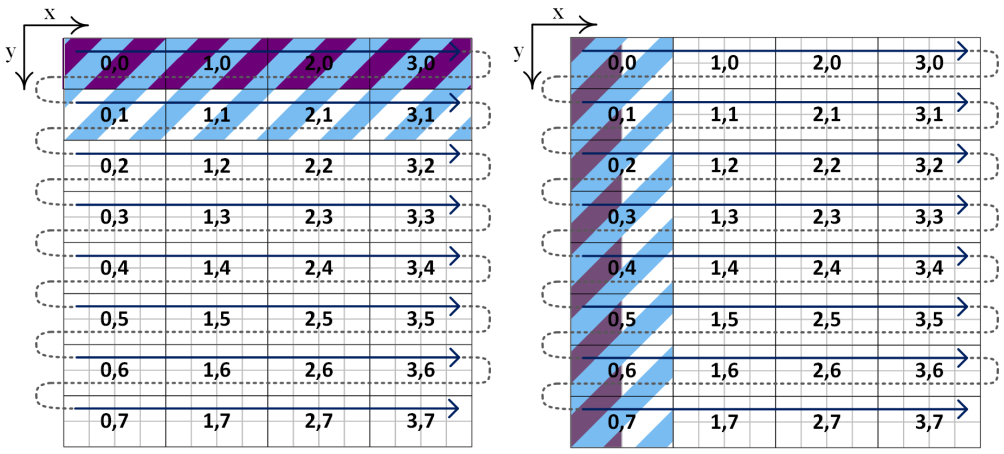
1. Number of concurrent transfers: running a kernel that reads or writes a remote memory we can obtain the effective peak remote memory bandwidth. If the kernel performs both read and write accesses we obtain the aggregated remote memory bandwidth.

2. Topology: crossing PCIe bridges to reach the destination GPU may increase the access latency and impact on the achieved bandwidth. *local* label indicates that no PCIe bridges are crossed, while *remote* indicates that one PCIe bridge is crossed. *host* indicates that remote data is stored in host memory.

Results in Figure 6.4 show the measured bandwidth on our two test systems. *copy* bars show the measured bandwidth using `cudaMemcpy` instead of remote memory accesses and *spec* show the maximum theoretical bandwidth offered by the interconnect. A single GPU can achieve the same memory bandwidth as bulk memory transfers by using remote memory accesses (>6 GBps for PCIe 2.0 and >12 GBps for PCIe 3.0). Write accesses exhibit 8-10% lower bandwidth than read accesses. When one GPU performs read and write remote accesses concurrently or two different GPUs concurrently perform

the same type of remote accesses (read or write), the measured aggregated memory bandwidth (>10 GBps for PCIe 2.0 and >18 GBps for PCIe 3.0) is higher than the peak memory bandwidth of a single GPU, although not twice. Crossing PCIe bridges imposes a noticeable overhead, especially for write accesses (>20% for reads, >30% for writes). Remote accesses to host memory exhibit a similar bandwidth than accesses to a GPU connected to the same PCIe bridge. The rest of experiments in this section are run on System B (PCIe 3.0).

Hypothesis 2 *Temporal distribution of remote memory accesses determine their effect on the kernel's performance. If all remote accesses are concentrated in the same phase of the kernel, it is more difficult to hide their costs as there is no other work that can be executed.*



(a) Rows (b) Columns

Figure 6.5: 2D kernel computation grids in which the 20% (blue) of the input and 10% (purple) of the output elements of the matrices are accessed remotely. In the configuration on the left, the first rows are remote. On the right, the first columns are remote. Labels indicate the x, y indices of the thread blocks within the computation grid. Arrows indicate the order in which thread blocks are executed.

The distribution of remote accesses is determined by the chosen data decomposition and the thread block scheduling policy. In current NVIDIA GPUs, thread blocks are scheduled for execution linearly following their identifier within the computation grid, from the lowest-order to the highest-order dimensions.

We execute a microbenchmark that uses a 2D computation grid in which each thread reads an element from an input matrix, performs 10 FLOPS on the

element and writes the result to an output matrix. A 20% of reads, and a 10% of writes are remote. We use the indices in the two different dimensions of the element being accessed by a thread in order to determine which accesses are remote, as shown in Figure 6.5. Two different data decompositions are emulated: (1) rows of the matrix are remote (Figure 6.5a), (2) columns of the matrix are remote (Figure 6.5b).

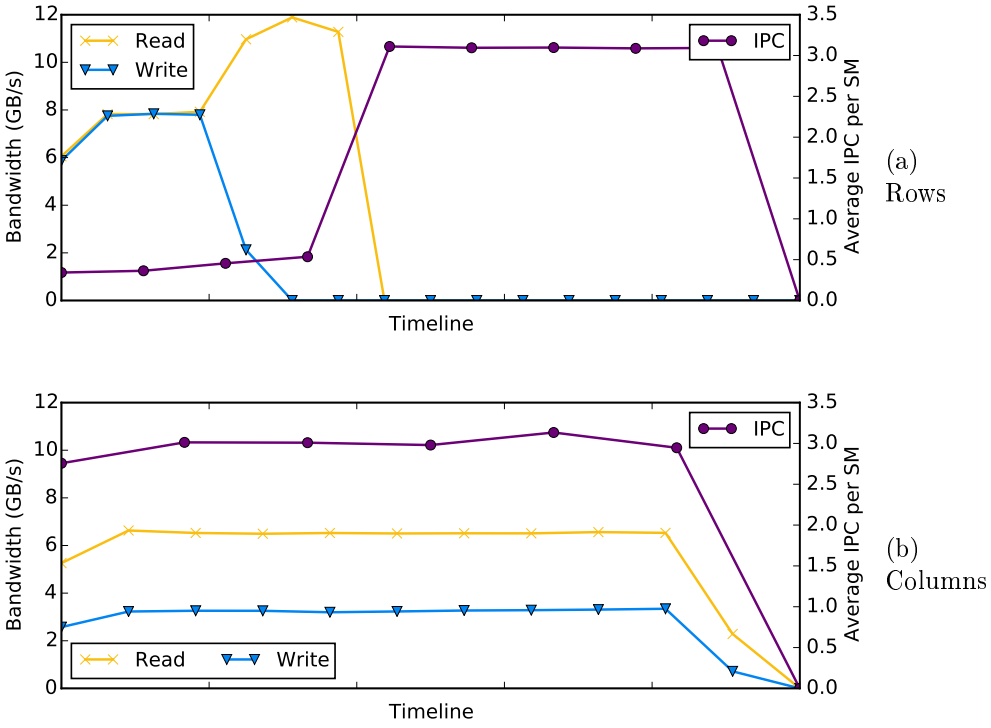
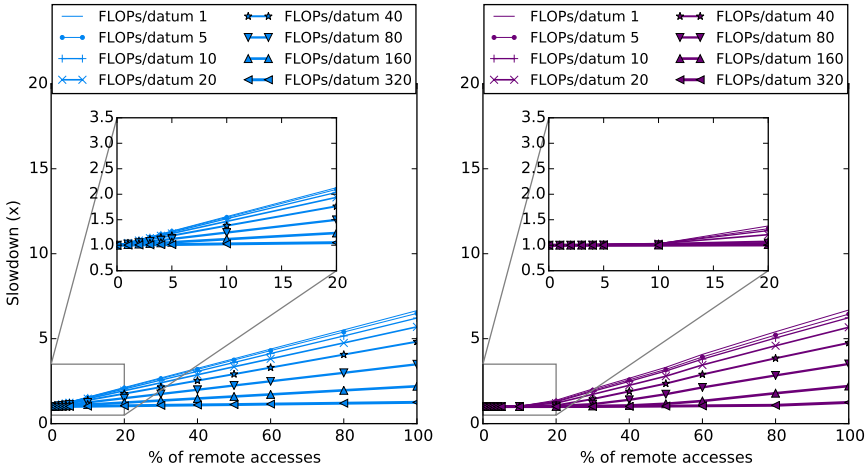


Figure 6.6: 2D kernel execution timeline in which the 20% of the input and 10% of the output elements of the matrices are accessed remotely. In the configuration on the top, the first rows are remote. On the bottom, the first columns are remote. Lines with circles indicate the average IPC per SM (right axis).

Figure 6.6 shows the distribution of remote accesses across the kernel execution time for the two different configurations. When rows are remote (top), we see that all read and write remote accesses are performed at the beginning of the kernel. This is because remote accesses are concentrated in a set of contiguous thread blocks that are executed at the same time. The achieved bandwidth is close to the peak bandwidth of the system measured in the previous experiment. When columns are remote (bottom), we see that remote accesses are spread through the whole kernel execution. In this case, thread blocks that perform remote accesses are executed concurrently with other thread blocks that do not. Thus, the requested remote bandwidth is



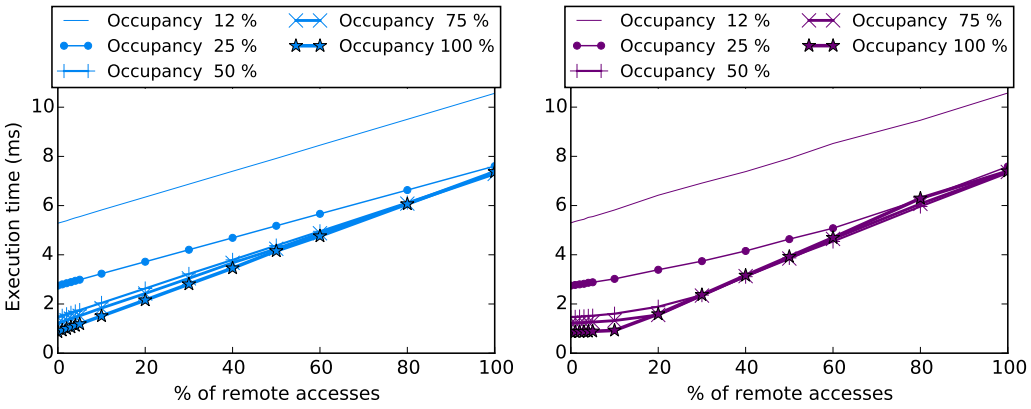
(a) Batch (b) Spread

Figure 6.7: Performance overhead imposed by remote accesses for different computation intensities. Figure on the left shows the overhead when all remote accesses are concentrated in time (thread blocks are scheduled together). Figure on the right shows the overhead when remote accesses are evenly distributed in time.

lower than the peak.
Lines with circles show the average IPC (i.e., instructions per cycle) per SM in the kernel. They show that requesting a remote memory bandwidth close to the peak (rows) hugely impacts on the performance of the kernel, while lower bandwidth requirements (columns) can be sustained through kernel execution with almost no impact on the performance.
These results confirm our hypothesis and indicate that the thread block scheduling policy must be taken into account when choosing the thread blocks to perform remote memory accesses.

Hypothesis 3 *Compute-bound computations suffer from less performance degradation than memory-bound computations due to remote memory accesses.*

We execute a number of GPU kernels that explore the following variables: (1) Amount of remote memory accesses relative to the total of memory accesses. (2) Amount of computations performed per input datum to encompass from completely compute-bound to memory-bound computation patterns. (3) Remote memory access distribution: *batch* or *spread*.
Figure 6.7 shows the overhead suffered by each of the GPU kernel configurations due to remote memory accesses. The overhead increases with the number of remote memory accesses (up to 7x slowdown), as expected. Com-



(a) Batch (b) Spread

Figure 6.8: Execution time for different remote accesses and SM occupancies. Figure on the left shows the execution time when all remote accesses are concentrated in time (thread blocks are scheduled together). Figure on the right shows the execution time when remote accesses are evenly distributed in time.

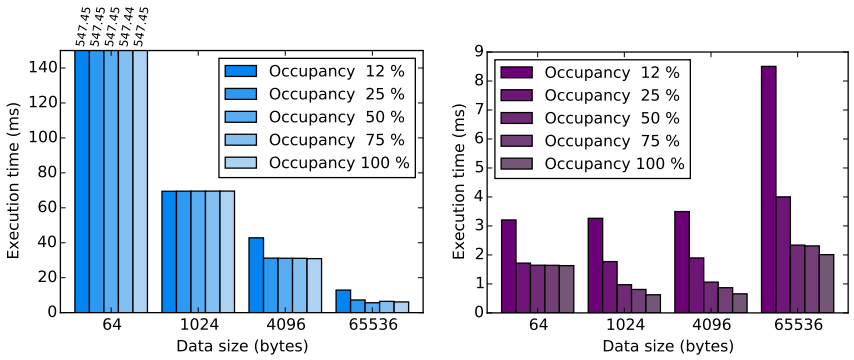
pute-bound computations (large FLOPS/datum values) are less affected than memory-bound computations. The distribution of remote memory accesses is key in order to hide their overhead. *batch* configuration (Figure 6.7a) shows a degradation of the performance that grows linearly with the amount of remote memory accesses. On the other hand, *spread* configuration (Figure 6.7b) shows in the zoomed section that values up to 10% of remote memory accesses result in less than 10% overhead for all computation intensities (including completely memory-bound computations). The overhead increases to 45% for the configuration with 20% of remote accesses and, for greater values, the overhead increases linearly, showing that the cost of remote accesses cannot be hidden any longer.
We can conclude that current GPUs and interconnects are able to hide the costs of 10% of remote accesses almost completely. Future interconnects such as NVLink will likely increase this number.

Hypothesis 4 *The GPU execution model helps hiding the costs of remote memory accesses by executing work from other warps.*

While Figure 6.7b already shows that the GPU execution model can hide the costs of a certain amount of remote memory accesses, we perform a more detailed analysis. Results in that figure assume maximum thread block concurrency (i.e., occupancy) in the SM. However, the amount of thread blocks that can be concurrently executed on an SM is limited by the amount

of resources used by each thread block. Many algorithms ported to GPU cannot reach the maximum occupancy, although most of them keep it high enough to be able to hide memory latency [85, 74].
We run the same GPU kernels used in the previous experiments, but we artificially modify the amount of scratchpad memory used per thread block, thus lowering the occupancy in the SMs. Results in Figure 6.8 show the execution time of the GPU kernel configuration that performs 10 FLOPs/datum for a different amount of remote memory accesses, and different SM occupancies and remote memory access distributions. Figure 6.8a (using *batch* distribution) shows that the execution time of the kernels increases linearly, and no occupancy is able to hide the costs of remote memory accesses. Figure 6.8b (using *spread* distribution) shows that occupancies starting at 25% are able to hide the costs of remote accesses and the execution time is lower than the *batch* distribution even for an 80% of remote memory accesses, thus confirming our hypothesis.

Hypothesis 5 *The overhead of remote accesses to small read/only data structures can be minimized using caching.*



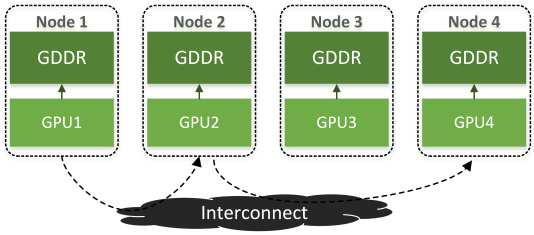
(a) Non-cached (b) Cached

Figure 6.9: Execution time for different sizes of the remotely accessed read-only data structure and different occupancies. Results for non-cached (left) and cached (right) accesses to the data structure. Note the different Y scales.

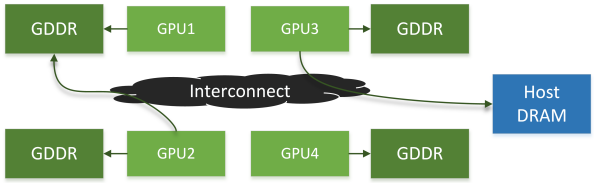
In the previous experiments, each thread reads different elements from either the remote or local allocations. However, oftentimes, all threads access the same elements of input data structures. For example, in the matrix-vector multiplication, every thread (or warp, depending on the implementation) accesses the whole input vector to compute the output element. Another example is the 2D convolution, in which each thread accesses all the elements

in the small convolution matrix and combines them with the input point and its neighbors to compute the output point.
We run again the microbenchmarks for the configuration in which 100% of input data accesses are remote and the code performs 10 FLOPS/datum. But this time we limit the size of the input data size so that there are elements that are read by several threads. Since the size of the computation grid is constant across kernel configurations, the smaller the input data structure, the more threads read each of its elements. Figure 6.9 shows the execution time of the benchmarks for different SM occupancies. Results show that when caching is not enabled, remote accesses to a small data structure impose a huge overhead and high occupancy values do not help hiding the latency. Interestingly, the smaller the data structure, the higher the overhead.
In conclusion, caching is effective for really small data structures as performance starts to degrade when the whole structure does not fit in the cache (the size of the R/O cache is 48 KB). The 64-byte configuration is also slower because it is smaller than the cache line size (required to achieve full bandwidth).

6.4 GPU-SM: towards shared memory multi-GPU programming



(a) Distributed memory model



(b) Shared memory model (GPU-SM)

Figure 6.10: Multi-GPU system models. Solid lines indicate access through load/store instructions; dashed lines represent bulk DMA transfers.

Given the results obtained in the previous section, we propose a set program-

ming practices to program multi-GPU systems as a shared memory machine.

6.4.1 Distributed memory vs shared memory

The current approach to multi-GPU programming is to use GPUs as nodes of a distributed system (Figure 6.10a). For example, in order to parallelize a kernel, programmers decompose computation and data and distribute them across GPUs. Data shared among different computation partitions is replicated, and outputs need to be gathered and, if they overlap, they need to be merged. In computations with data dependencies across computation partitions, data generated in one GPU might need to be explicitly transferred to other GPU memories before next kernels can proceed. These copies perform bulk DMA transfers across the interconnect. However, these communications can introduce a large overhead, and techniques to overlap computation with communication are commonly used, at the cost of increased code complexity. We propose changing the perspective of how a multi-GPU system is programmed by looking at the GPUs and their memories as if they were a shared memory NUMA system (Figure 6.10b), which we call GPU-SM. Any GPU can access all the memories using regular load/store operations, thanks to the UVAS and remote memory access capabilities introduced in Chapter 2. Thus, programmers can freely allocate data in any of the memories. Nevertheless, the overhead imposed by remote memory accesses can be large, and data should be placed in such a way that they are minimized.

6.4.2 Writing code for GPU-SM

When a GPU kernel is launched, a grid of thread blocks is instantiated and a hardware scheduler distributes them for execution among the SMs in the GPU. Since GPUs can now access all memories in the system, executing a GPU kernel across all the GPUs in the system could be as simple as aggregating all their SMs and memories. Thus, the scheduler would just need to distribute the thread blocks among a larger amount of SMs. However, while a GPU can contain several SMs, it is exposed as a single compute unit, and current schedulers cannot issue thread blocks to different GPUs. Hence, programmers have to decompose the computation grid into partitions and launch each partition on a different GPU.
The computation grid is a multidimensional space of thread blocks of up to 3 dimensions ($grid_x \times grid_y \times grid_z$). Programmers can decompose the grid on any of their dimensions (or a combination of them). The dimensions being decomposed determine how data structures must be partitioned and distributed. This is because programmers usually apply affine transformations

to the block and thread indices to compute the indices that are used to access data structures. As a result, threads that belong to blocks with contiguous identifiers in one dimension tend to access elements that are contiguous in the same or a different dimension of the referenced data structure.
We now discuss important trade-offs when using GPU-SM.

Computation decomposition and its impact on the remote memory access pattern

When thread blocks access mostly non-overlapping regions of a data structure, the data structure can be decomposed in such a way that a data partition is predominantly accessed by a single partition of the computation grid. If there is overlap between accessed data regions, the chosen dimensions for the computation grid decomposition determine the pattern of the remote memory accesses. This is caused by the thread block scheduling policy in the GPU. For example, consider a 2D stencil computation example, that creates a 2D computation grid in which each thread uses its linear indices in X and Y dimensions to access the matrix, reading an input element and the k neighboring elements in the X and Y dimensions. This computation pattern creates a *halo* of elements that are needed by the threads in the edges of the thread block. Thus, neigboring thread blocks access the elements in this halo of points and, therefore, the thread blocks in the boundaries of the computation partitions will need to access data that is located in a different memory. Depending on the dimension being decomposed, accesses to this data arrive with different distributions. When the X dimension is decomposed, some columns of the input matrix are accessed remotely. Therefore, accesses will be performed by thread blocks that are evenly distributed in the kernel execution time. On the other hand, when the Y dimension is decomposed, some rows of the input matrix are accessed remotely, which creates a single big batch of remote access and is more likely to harm the performance of the kernel. As we have seen in Section 6.3, decomposing the X dimension would be better in this case.
A better approach would let programmers choose between different scheduling policies so that implementing different computation and data decompositions would not be required. We expect that future GPUs will offer the possibility to tune some of their internal mechanisms, such as scheduling, to allow experts to better exploit the capabilities of the hardware.

Replication versus caching

Another common memory access pattern is produced when every thread in the kernel traverses a whole input data structure. The distributed approach would use replication to eliminate the need for remote accesse at all, at the cost of increased host code complexity. But, as we have seen, small input data structures can be efficiently cached using the `__ldg` intrinsic. Therefore, replication can be avoided in kernels in which input data structures fit in the L1 R/O cache (48 KB in Kepler GPUs). If other data structures in the kernel might also benefit from `__ldg`, we should carefully choose which ones are cached.

Output data structures

Replicating output data structures implies that copies need to be merged after kernel execution in order to have a consistent version in all GPUs. This step imposes an overhead that, in most the cases, is larger than the costs of using remote updates. Moreover, replication limits the size of the problem that can be handled. Furthermore, code complexity greatly increases in order to implement merging efficiently. Fortunately, GPUs perform better with gather memory access patterns [85], which minimize the number of write operations to the GPU memory. Therefore, replication of output data should only be used if remote writes limit kernel's performance.

Memory fences

GPUs implement several memory fence instructions that work with different granularities: block-level, GPU-level, and system-level. In the first case, the calling thread waits until all its writes to memory are visible to all threads in the thread block. The other two extend the visibility of the writes to the threads of all the thread blocks in the GPU and the system (all GPUs), respectively. GPU-level and system-level fences are provided to enable synchronization across thread blocks in a kernel or across kernels launched on different GPUs. In the typical distributed system model, a GPU only accesses data stored in its memory. In the shared memory model, computations decomposed to be executed across several GPUs may access data stored remotely. Thus, if the kernel uses GPU-level memory fences, they need to be upgraded to system-level memory fences since all the kernel partitions executed on each GPU may work on the same data.

6.4.3 Current limitations

Virtual memory

CPU-based shared memory systems use VM mechanisms such as page-fault handling to transparently detect the processors that request data. Thus, allocation [29, 30], replication [38] and migration [72] policies can be transparently implemented to minimize the amount of remote memory accesses. Unfortunately, GPUs do not implement such VM mechanisms yet and programmers perform data placement manually. However, vendors have announced true VM capabilities such as page-fault for future GPUs, which will allow for automatic memory placement policies.
The CUDA API does not provide the functionality to allow programmers manage the virtual addresses in the UVAS. This feature would enable transparent mapping of contiguous pages to alternate physical memories which, in turn, would make data distribution completely transparent to the kernel code[1]. Currently, different allocations must be used to distribute data across different GPU memories, and the code must be aware of the different allocations and choose the appropriate pointer. Nevertheless, we consider that this limitation in the API will be fixed in the future.

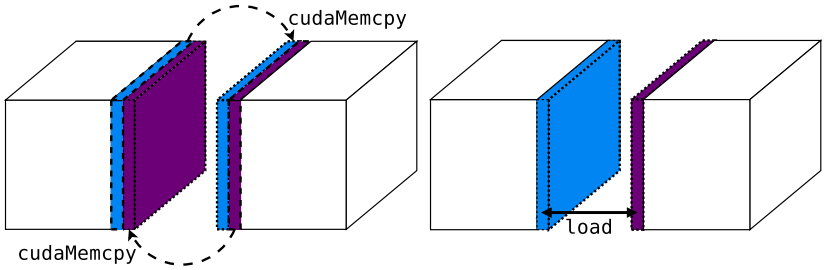
Atomics

GPUs support atomic operations, which are implemented in the L2 cache. Since regular accesses to remote data cannot be cached, data resides in the GPU cache hierarchy whose physical memory contains the accessed data, only. This limitation simplifies the support of atomic instructions across GPUs as they only need to be forwarded to the GPU that owns the data. PCIe 3.0 does support atomic operations but current NVIDIA GPUs do not forward atomic operations to remote GPUs. Thus, we cannot port kernels that use atomic operations to the GPU-SM model using current GPUs.

6.5 Implementation and performance evaluation of GPU-SM applications

In order to evaluate the performance, we have ported two applications from its original distributed model implementation to GPU-SM.

[1]Using page granularity for data decomposition can also introduce unnecessary remote accesses. This problem is disussed in more detail in Chapter 7.



(a) Distributed memory (b) GPU-SM

Figure 6.11: Inter-domain data dependences for FDTD.

6.5.1 FDTD

The first application is a simplified version of the guiding example of this thesis, RTM, which is described in Section 6.2.3. We discuss the differences between the distributed and shared memory implementations.

Distributed memory implementation

In this implementation, the halo data that is shared between domains is replicated in each of the GPU memories (Figure 6.11a). This implies that GPUs exchange data between simulation steps and, therefore, exchange efficiency is critical to the scalability of such applications.
The GPU kernel uses a 2D computation grid that is mapped on the front XY plane of the volume. Each thread iterates through all the planes, computing a single output element for each plane. The implementation used in our tests is based on [68], which uses register tiling in the Z dimension of the array, and shared memory to reuse the elements read by neighboring threads in the X and Y dimensions. Due to the length of the optimized version of the kernel, we show a simplified version in Listing 6.1. The same GPU kernel can be used in both single- and multi- GPU versions, as each computation partition accesses data stored in its own GPU memory.
Listing 6.2 shows an optimized version of the host code that decomposes the simulated domain on its X dimension. It is much more complex than the single-GPU version and it heavily relies on CUDA abstractions such as streams and events in order to overlap computation and data transfers. Each iteration of the simulation is divided in three main phases.

1. Compute boundaries (lines 16–33): a kernel is launched to compute the points in the boundaries, so that they can be communicated as soon as possible. Kernels are queued into different streams so they can be concurrently executed on the GPU.

2. Compute center (lines 35–41): a kernel is launched on a third stream to compute the rest of the points in the subdomain.

3. Exchange boundaries (lines 43–58): then, memory transfers are queued in the two first streams in order that they start as soon as the kernels that compute the points finish.

These steps are repeated for each GPU in the system. Each operation is tracked using an event. Event barriers are also pushed to streams before each of the listed steps thus preventing operations of the current time step to start before the operations launched in the previous time step have consumed or produced the required data.

GPU-SM implementation

Using GPU-SM, halo data does not need to be replicated in GPU memories and it is accessed through remote accesses (Figure 6.11b). Since CUDA currently does not allow us to transparently distribute the volumes by mapping pages on alternate GPUs, we use one allocation per GPU and the kernel is modified to be aware of the different allocations. Thus, threads that compute the elements in the boundaries use the pointers to the remote allocations (lines 13 and 26 in Listing 6.3). Without this restriction the kernel code would be the same as in Listing 6.1. On the other hand, the host code is much simpler than the distributed version since there are no data transfers, a single kernel launch computes the whole domain (lines 27–30 in Listing 6.4), and only one stream per GPU is used.

Performance analysis

Figure 6.12 shows the speedup achieved by running the different implementations of the FDTD benchmark on 4 GPUs, compared to the original implementation on a single GPU. In the configuration with the smallest volume, the Z decomposition is the best one for both distributed and GPU-SM implementations. This is because the computation grid is very small (32×4) and it is further reduced (32 blocks per GPU) when the computation is decomposed on the X or Y dimensions, which leads to the underutilization of the SMs. When decomposed on the Z dimension, threads iterate over a smaller number of planes, but the computation grid it is not decomposed, and more thread blocks can run on each GPU concurrently. Bigger volume sizes produce higher speedups for all the configurations due to a larger number of thread blocks and a lower relative amount of remote accesses. Z

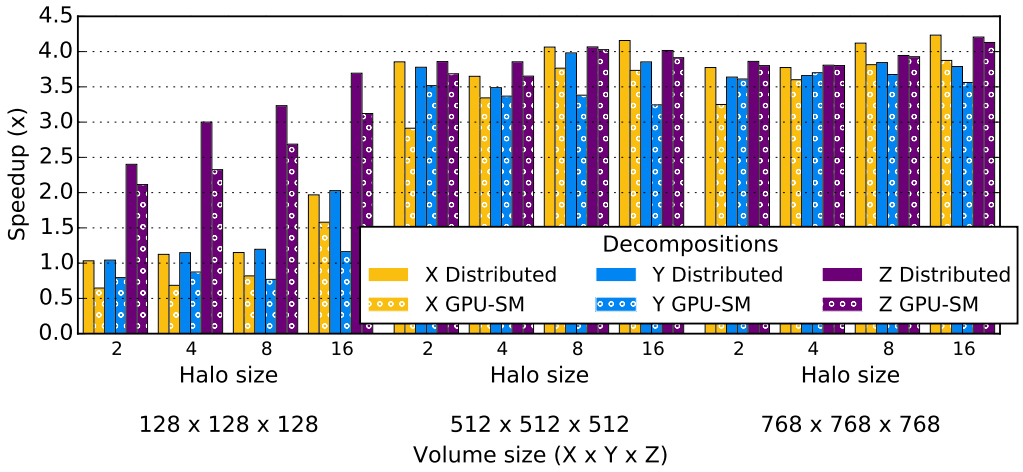
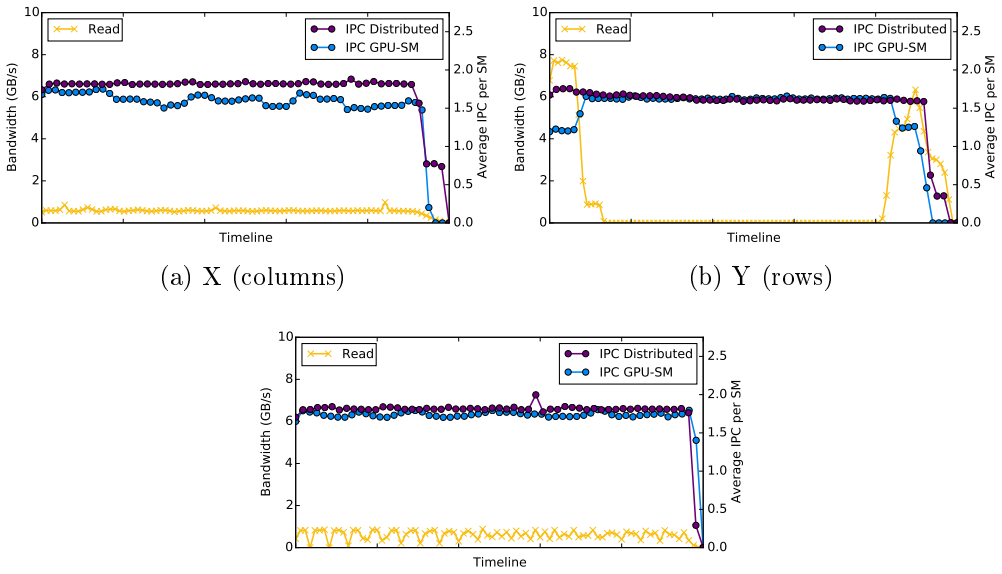


Figure 6.12: FDTD: speedups of the multi-GPU implementations for 4 GPUs compared to the original implementation on a single GPU. Results for different halo sizes and volume sizes.

remains as the best decomposition for GPU-SM implementations, and provides $> 3.60\times$ speedups for the $512 \times 512 \times 512$ volume and $> 3.80\times$ for the $768 \times 768 \times 768$. Compared to the distributed implementation, the overhead due to remote accesses is always lower than 8% for the two bigger volumes. Figure 6.13 shows the execution timelines of the benchmark configuration with $768 \times 768 \times 768$ volume size, 16 halo size, for the X, Y and Z decompositions. The timelines show the remote memory access throughput and the average IPC per SM of the kernel for both the distributed and the GPU-SM versions. Results in Figure 6.12 indicate that the best decomposition for this configuration is Z, followed by X and Y. Note that the X and Z decompositions exhibit higher IPC than Y even for the distributed implementation, due to better spatial locality. The Y decomposition (rows) suffers a sharp drop in IPC due to the high remote access throughput (up to 8 GBps) of the thread blocks that execute at the beginning and the end of the kernel. The rate of remote accesses is constant across the execution of the X decomposition, while the Z decomposition shows a jagged pattern. The IPC of X degrades over time (because thread blocks performing remote accesses accumulate), while in Z all thread blocks perform remote accesses but only for a small period of time (the few first/last planes in the domain) that can be hidden with the execution of other thread blocks.



(a) X (columns)

(b) Y (rows)

(c) Z (planes)

Figure 6.13: FDTD: execution timeline for different decompostion configurations. Line with crosses indicate the achieved remote bandwidth. Lines with circles indicate the average IPC per SM.

6.5.2 Image filtering

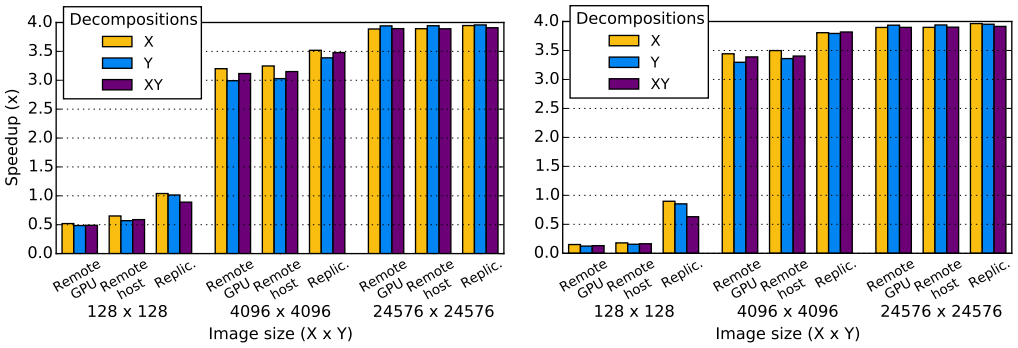
The second evaluated application is Image Filtering, as described in Section 6.2.3.

Distributed memory implementation

The traditional implementation of convolution for multiple GPUs replicates the halos of the partitions of the input image in each GPU. The convolution matrix is replicated in all GPU memories. The output image does not require replication because computation partitions write in non-overlapping regions.

GPU-SM implementation

In GPU-SM, thanks to shared memory, halo data for the input image partitions can be remotely accessed. With respect to the convolution matrix, it cannot be decomposed because it is fully accessed by every thread in each computation partition. We study two possibilities: (1) replicating it like in the distributed implementation, (2) storing it in a single GPU or in host memory and use caching, as it is small enough to fit in the L1 R/O cache.



(a) 3×3 convolution (b) 5×5 convolution

Figure 6.14: Image filtering: speedups of the multi-GPU GPU-SM implementation for 4 GPUs compared to the original implementation on a single GPU. Results provided for different convolution matrix sizes and locations, and image sizes.

Performance analysis

Figure 6.14 shows the speedup achieved by running the GPU-SM implementation of the convolution kernel on 4 GPUs. Like in the FDTD benchmark, the smallest input data set is too small to benefit from multi-GPU execution. The bars show the results for the configurations that use the L1 R/O cache. Not using this cache introduced up to 1000× slowdowns when accessed remotely as each thread performs reads every element in the matrix (9 or 25 in our benchmarks), and each access triggers a remote memory access. Further, using the R/O cache is also a 35% faster than the regular cache hierarchy when the convolution matrix is replicated. Results for the medium-size image indicate that there is around a 10% performance loss when not using replication and resorting to caching to access the convolution matrix for both the 3×3 and 5×5 convolution matrix sizes (storing it in host memory instead of a GPU memory improves the results by 2%). This is because all the thread blocks in all SMs (> 100) are stalled at the beginning of the kernel until each of the SMs brings the matrix to its L1 R/O cache and, therefore, this cost cannot be hidden. Having a L2 shared R/O cache would help minimizing the performance overhead as only the first SM would need to bring the matrix and the rest of SMs could access cached data. For larger matrix sizes these initial costs are negligible and we then achieve linear speedups.

```
1  template <typename T, unsigned Halo>
2  __global__ void
3  stencil(T* out, const T* in,
4          unsigned cols, unsigned rows, unsigned planes)
5  {
6    int k = threadIdx.x + blockIdx.x * blockDim.x + Halo;
7    int j = threadIdx.y + blockIdx.y * blockDim.y + Halo;
8
9    if ((k < Halo + cols) && j < (Halo + rows)) {
10     for (int p = Halo; p < planes + Halo; ++p) {
11       T c = IN(k, j, p);
12       for (int s = 1; s <= Halo; ++s) {
13         T left   = in[IDX_3D(k-s, j, p)];
14         T right  = in[IDX_3D(k+s, j, p)];
15         T top    = in[IDX_3D(k, j-s, p)];
16         T bottom = in[IDX_3D(k, j+s, p)];
17         T back   = in[IDX_3D(k, j, p-s)];
18         T front  = in[IDX_3D(k, j, p+s)];
19
20         c += 3.f * (left + right) +
21              2.f * (top + bottom) +
22              1.f * (back + front);
23       }
24       out[IDX_3D(k, j, p)] = c;
25     }
26   }
27 }
```

Listing 6.1: FDTD: distributed implementation (simplified version of the kernel).

6.6 Summary

In this chapter we have shown that the remote memory access mechanism enables easier multi-GPU programming. While PCIe offers limited bandwidth compared to GPU memories, GPU's ability to hide long latency accesses allows it to tolerate a moderate amount of remote accesses. More precisely, GPUs are able to hide the costs of a 10% of remote memory accesses if the kernels produce high GPU core occupancy and the accesses are spread through the whole kernel execution. We have also shown that shared memory implementations of FDTD and image filtering computations deliver a performance comparable to their highly-optimized distributed counterparts, while being simpler and much more concise. Future interconnects promise higher bandwidths that will open the GPU-SM model to a broader range of applications.
This work has also identified hardware improvements that may have the potential to hide the costs of remote accesses.

- Increasing the number of in-flight warps per GPU core would allow to tolerate a higher number of remote accesses.

- Giving controls to programmers to change the thread block scheduling

```
1  struct work_descriptor {
2    float *in, *out;
3    cudaStream_t stream[NUM_STREAMS];
4    cudaEvent_t events_A[NUM_EVENTS];
5    cudaEvent_t events_B[NUM_EVENTS];
6    cudaEvent_t *events_prev = events_A;
7    cudaEvent_t *events_cur = events_B;
8    bool has_left_neigh;
9    bool has_right_neigh;
10   unsigned planes;
11 };
12 void do_rtm(work_descriptor wd[NUM_GPUS])
13 {
14   for (int t = 0; t < TIME_STEPS; ++t) {
15     for (int gpu = 0; gpu < NUM_GPUS; ++gpu) {
16       // 1a. Compute right boundary
17       if (wd[gpu].has_left_neigh)
18         cudaStreamWaitEvent(wd[gpu].stream[EXEC_R], wd[gpu-1].events_prev[COMM_R]);
19
20       cudaStreamWaitEvent(wd[gpu].stream[EXEC_R], wd[gpu].events_prev[EXEC_M]);
21       launch_stencil(wd[gpu].in, wd[gpu].out,
22                      halo + wd[gpu].planes - 2 * halo, halo * 3, // offset, size
23                      wd[gpu].stream[EXEC_R]);
24       cudaEventRecord(wd[gpu].events_cur[EXEC_R], wd[gpu].stream[EXEC_R]);
25       // 1b. Compute left boundary
26       if (wd[gpu].has_right_neigh)
27         cudaStreamWaitEvent(wd[gpu].stream[EXEC_L], wd[gpu+1].events_prev[COMM_L]);
28
29       cudaStreamWaitEvent(wd[gpu].stream[EXEC_L], wd[gpu].events_prev[EXEC_M]);
30       launch_stencil(wd[gpu].in, wd[gpu].out,
31                      0, halo * 3, // offset, size
32                      wd[gpu].stream[EXEC_L]);
33       cudaEventRecord(wd[gpu].events_cur[EXEC_L], wd[gpu].stream[EXEC_L]);
34
35       // 2. Compute center
36       cudaStreamWaitEvent(wd[gpu].stream[EXEC_M], wd[gpu].events_prev[EXEC_L]);
37       cudaStreamWaitEvent(wd[gpu].stream[EXEC_M], wd[gpu].events_prev[EXEC_R]);
38       launch_stencil(wd[gpu].in, wd[gpu].out,
39                      halo, wd[gpu].planes, // offset, size
40                      wd[gpu].stream[EXEC_M]);
41       cudaEventRecord(wd[gpu].events_cur[EXEC_M], wd[gpu].stream[EXEC_M]);
42
43       // 3a. Exchange right boundary
44       if (wd[gpu].has_right_neigh) {
45         cudaStreamWaitEvent(wd[gpu].stream[COMM_R], wd[gpu].events_cur[EXEC_R]);
46         copyAsync(wd[gpu+1].out, wd[gpu].out,
47                   halo + wd[gpu].planes - halo, halo, // offset, size
48                   wd[gpu].stream[COMM_R]);
49         cudaEventRecord(wd[gpu].events_cur[COMM_R], wd[gpu].stream[COMM_R]);
50       }
51       // 3b. Exchange left boundary
52       if (wd[gpu].has_left_neigh) {
53         cudaStreamWaitEvent(wd[gpu].stream[COMM_L], wd[gpu].events_cur[EXEC_L]);
54         copyAsync(wd[gpu-1].out, wd[gpu].out,
55                   halo, halo, // offset, size
56                   wd[gpu].stream[COMM_L]);
57         cudaEventRecord(wd[gpu].events_cur[COMM_L], wd[gpu].stream[COMM_L]);
58       }
59     }
60     for (int gpu = 0; gpu < NUM_GPUS; ++gpu) {
61       swap(wd[gpu].in, wd[gpu].out);
62       swap(wd[gpu].events_prev, wd[gpu].events_cur);
63     }
64   }
65 }
```

Listing 6.2: FDTD: distributed implementation (host).

```
1  template <typename T, unsigned Halo>
2  __global__ void
3  stencil(T* out, const T* in, const T* in_left, const T* in_right,
4          unsigned cols, unsigned rows, unsigned planes)
5  {
6    int k = threadIdx.x + blockIdx.x * blockDim.x + Halo;
7    int j = threadIdx.y + blockIdx.y * blockDim.y + Halo;
8
9    if (k < cols && j < rows) {
10     for (int p = Halo; p < planes + Halo; ++p) {
11       T c = in[IDX_3D(k, j, p)];
12       for (int s = 1; s <= Halo; ++s) {
13         T left = (in_left && k-s < 0)? in_left[IDX_3D(cols + (k-s), j, p)]:
14                                              in[IDX_3D(k-s, j, p)];
15
16         T right = (in_right && k+s >= cols)? in_right[IDX_3D((k+s) - cols, j, p)]:
17                                                    in[IDX_3D(k+s, j, p)];
18
19         T top    = in[IDX_3D(k, j-s, p)];
20         T bottom = in[IDX_3D(k, j+s, p)];
21         T back   = in[IDX_3D(k, j, p-s)];
22         T front  = in[IDX_3D(k, j, p+s)];
23
24         c += 3.f * (left + right) +
25              2.f * (top + bottom) +
26              1.f * (back + front);
27       }
28       out[IDX_3D(k, j, p)] = c;
29     }
30   }
31 }
```

Listing 6.3: FDTD: GPU-SM implementation (simplified version of the kernel).

```
1  struct work_descriptor {
2    float *in, *out;
3    cudaStream_t stream;
4    cudaEvent_t event_prev;
5    bool has_left_neigh;
6    bool has_right_neigh;
7    unsigned planes;
8  };
9
10
11
12 void do_rtm(work_descriptor wd[NUM_GPUS])
13 {
14   for (int t = 0; t < TIME_STEPS; ++t) {
15     for (int gpu = 0; gpu < NUM_GPUS; ++gpu) {
16       float *in_left = nullptr;
17       float *in_right = nullptr;
18       if (wd[gpu].has_left_neigh) {
19         in_left = wd[gpu-1].in;
20         cudaStreamWaitEvent(wd[gpu].stream, wd[gpu-1].event_prev);
21       }
22       if (wd[gpu].has_right_neigh) {
23         in_right = wd[gpu+1].in;
24         cudaStreamWaitEvent(wd[gpu].stream, wd[gpu+1].event_prev);
25       }
26       cudaStreamWaitEvent(wd[gpu].stream, wd[gpu].event_prev);
27       launch_stencil(wd[gpu].in, in_left, in_right, wd[gpu].out,
28                      wd[gpu].has_left_neigh  ? 0 : halo, // offset
29                      wd[gpu].planes + (wd[gpu].has_right_neigh ? 0 : halo), // size
30                      wd[gpu].stream);
31     }
32
33     for (int gpu = 0; gpu < NUM_GPUS; gpu++) {
34       cudaEventRecord(wd[gpu].event_prev, wd[gpu].stream);
35       swap(wd[gpu].in, wd[gpu].out);
36     }
37   }
38 }
```

Listing 6.4: FDTD: GPU-SM implementation (host).

policy, would allow them to distribute remote accesses along the whole kernel execution, without the need to change the code of the application (i.e., data and computation distribution).

- Adding a L2 read-only cache would allow a GPU core to reuse data structures (especially small ones) that have been cached by a different GPU core.

Chapter 7

Automatic Multi-GPU Execution

In the previous chapter we show that remote access is a viable mechanism to implement shared memory programming on GPUs. However, the costs of only a limited amount of remote memory accesses can be hidden by the GPU architecture and execution model. Therefore, programmers are still in charge to carefully distribute data structures so that the performance is not affected. They also need to decide when to use replication.
In this chapter we present AMGE: an auto-parallelization programming framework for multi-GPU systems that relieves programmers from these duties.

7.1 AMGE overview

AMGE (Automatic Multi-GPU Execution) is a programming framework that decomposes and distributes GPU kernels and data to be collaboratively executed on all the GPUs in the system. We implement AMGE using C++ and CUDA, but it can be extended to other languages. Figure 7.1 shows the components in AMGE and how they interact with the hardware. AMGE aggregates the GPU resources in the system and presents them as a *single virtual GPU*. Thus, programmers are relieved from the burden of decomposing the problem and explicitly managing several GPUs.
The AMGE compiler is a source-to-source compiler, that analyzes the CUDA kernels in the program to detect their array access patterns and store this information in the program executable. It also generates optimized kernel versions for the possible array decompositions. We argue that the utiliza-

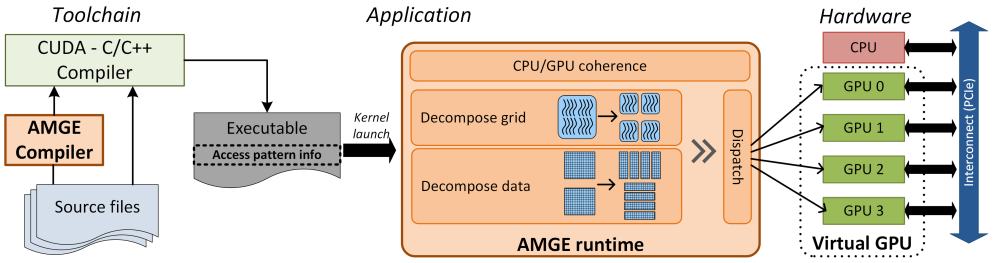


Figure 7.1: Overview of AMGE components. The compiler extracts array access pattern information and stores it in the program binary. The runtime system uses this information to decompose and distribute computation and data across the GPUs in the system. In this example, the system is composed of a single CPU and 4 GPUs, connected through a PCIe interconnect.

tion of the array dimensionality information is paramount in order to efficiently exploit multi-GPU systems. However, CUDA is an extension of the C/C++ languages, which do not provide data types with such information; programmers typically flatten the multi-dimensional arrays into 1D arrays and linearize the dimension indices in each array reference. It is practically difficult, if not infeasible, for static analysis to reliably recover the dimensionality information once the accesses have been flattened. AMGE provides a new data type for multi-dimensional arrays that makes this information available to the compiler. Details on the implementation of the data type and the generation of optimized kernel versions are discussed in Section 7.3. The other key feature of AMGE is the utilization of *remote memory accesses* between GPUs [5]. On each reference to the array, the underlying implementation determines whether the element being referenced is hosted in the memory local to the GPU executing the code or on a different GPU. References from a GPU to parts of the array stored in different GPU memories are handled using remote memory accesses. This approach ensures correctness regardless of the chosen GPU computation and data distribution configuration and removes the requirement for the compiler analysis to unequivocally determine the bounds of the memory range accessed by a computation partition.

However, remote accesses can impose performance overheads and they must be minimized. On each kernel call, *the AMGE runtime* determines the best computation and array decompositions using the information generated by the compiler, and distributes them across all GPUs in the system.

```
1  void sgemm(ndarray<float, 2, storage::cmo> C, ndarray<float, 2, storage::cmo> A,
2             ndarray<float, 2> B)
3  {
4     float partial[SGEMM_TILE_N];
5     __shared__ float b_tile_sh[SGEMM_TILE_HEIGHT][SGEMM_TILE_N];
6     for (int i = 0; i < SGEMM_TILE_N; i++) partial[i] = 0.0f;
7
8     int mid = threadIdx.y * blockDim.x + threadIdx.x;
9     int row = blockIdx.x * (SGEMM_TILE_N * SGEMM_TILE_HEIGHT) + mid;
10    int col = blockIdx.y *  SGEMM_TILE_N + threadIdx.x;
11
12    for (int i = 0; i < A.get_dim(1); i += SGEMM_TILE_HEIGHT) {
13       b_tile_sh[threadIdx.y][threadIdx.x] = B(i + threadIdx.y, col);
14       __syncthreads();
15       for (int j = 0; j < SGEMM_TILE_HEIGHT; ++j) {
16          float a = A(row, i + j);
17          for (int k = 0; k < SGEMM_TILE_N; ++k)
18             partial[k] += a * b_tile_sh[j][k];
19       }
20       __syncthreads();
21    }
22    for (int i = 0; i < SGEMM_TILE_N; i++)
23       C(row, i + by * SGEMM_TILE_N) = partial[i];
24 }
```

Listing 7.1: Multi-GPU `sgemm` GPU code with AMGE.

Memory model

Arrays are transparently decomposed and/or replicated before each kernel call. Input arrays can be replicated at the cost of additional space and data transfers, but AMGE never replicates output arrays. Replicated output arrays requires additional coherence management to merge partial modifications on different GPUs. Previous works [57, 61] have to transfer all copies to the host memory for a merging step after every kernel call, imposing a large performance overhead in many workloads. Moreover, replicating output arrays prevents distributing codes with atomics or memory fences. AMGE always distributes output arrays across GPU memories instead, and relies on remote memory accesses to guarantee that they are available to all GPUs.
AMGE implements the ADSM model [45] to allow arrays to be used both by host and GPU code. The runtime only transfers arrays between CPU and GPU memories when needed.

7.1.1 An example: matrix multiplication

Code programmed to run on a single GPU requires only minor modifications to use AMGE. Listing 7.1 shows the GPU code of a single-precision floating point matrix-matrix multiplication computation [44] (i.e., `sgemm`) using AMGE. A and C matrices are stored in *column major* order, and B in *row major* order, for optimal performance. The highlighted text shows the mod-

```
1  // Initialize A and B in the host code
2  ndarray<float, 2, storage::cmo> A;
3  ndarray<float, 2>               B;
4
5  read_array("A.dat", A);
6  read_array("B.dat", B);
7
8  ndarray<float, 2, storage::cmo> C(A.get_dim(1), B.get_dim(0));
9  // Computation grid size
10 dim3 block(MATRIXMUL_TILE_N, SGEMM_TILE_HEIGHT);
11 dim3 grid(C.get_dim(1)/(SGEMM_TILE_N * SGEMM_TILE_HEIGHT),
12           C.get_dim(0)/SGEMM_TILE_N);
13 // Kernel launch. A, B and C are used in the GPU code
14 sgemm<<<grid, block>>>(C, A, B);
15 // Write results for C into a file
16 write_array("C.dat", C);
```

Listing 7.2: Multi-GPU `sgemm` host code with AMGE.

ifications performed to the original code. The only additional programming requirement for the kernel to be automatically decomposed is the array data type (lines 1-2), and its associated indexing routines (lines 12, 13, 16 and 23). The data type is implemented by the `ndarray<T, Dims, Storage>` C++ class template, where `T` is the type of the elements, `Dims` is the number of dimensions of the array, and `Storage` is an optional parameter (`storage::cmo` or `storage::rmo`) that defines the storage type. By default, the C/C++ *row major* order(i.e., `storage::rmo`) storage convention is used. The kernel uses a 2D computation grid, in which each thread block computes a 2D tile of C by traversing A and B on their X and Y dimensions, respectively. The compiler detects these patterns and stores them in the program executable (*access pattern info* in Figure 7.1).
Listing 7.2 shows the CPU code of the `sgemm` computation. First, `float` input matrices `A` and `B` are declared in lines 2 and 3. The bounds of each dimension of the array are defined at run-time. The AMGE runtime intercepts the kernel call and uses the information generated by the compiler to decompose the matrices, and distribute both computation and data across all the GPUs. Note that `ndarray` objects can be passed both to CPU and GPU routines, making explicit data transfers between host and GPU memories unnecessary.

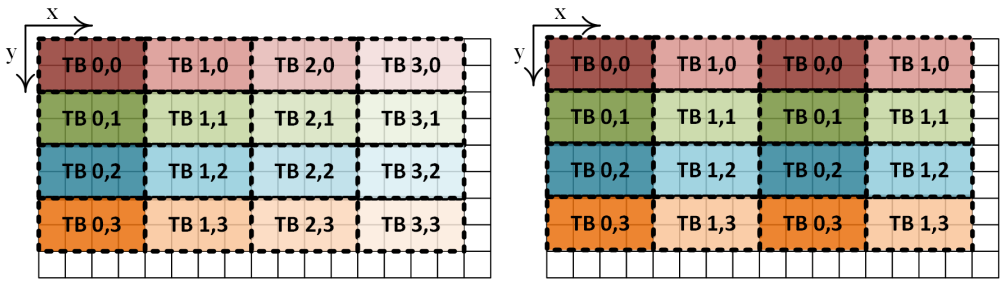
7.2 Computation and data distribution

AMGE decomposes GPU kernels at thread block boundaries. We choose this decomposition granularity because threads within a thread block share resources (e.g., shared memory), and perform barrier synchronization operations. Hence, all threads within a thread block must be executed in the same compute core of the same GPU. However, the GPU programming model

guarantees that there are no data dependences across thread blocks within a kernel and, therefore, they can execute independently.
In CUDA, programmers specify a computation grid that is a multidimensional space $grid_x \times grid_y \times grid_z$ of thread blocks, similar to the iteration space in loop nests. Each thread block has a unique identifier $0 \leq block_x, block_y, block_z < grid_x, grid_y, grid_z$ within the computation grid. The AMGE runtime decomposes the computation grid so that it can be executed on several GPUs. In CUDA and OpenCL, the computation grid is canonical and rectangular. Thus, the computation grid can be uniformly decomposed into partitions along one or several of its dimensions.
AMGE uses compiler analysis to generate array access pattern information for all the GPU kernels in the program. This information is later used by the runtime system to try to place on each GPU the data accessed by the computation partition assigned to it, in order to minimize the number of remote memory accesses.

7.2.1 Compiler analysis

The AMGE compiler analyzes all array references in the kernel. Each reference contains one index for each of the dimensions of the array, allowing the AMGE compiler to detect the individual access pattern in each dimension. This is in contrast to previous works [57, 61] that treat all arrays as one dimensional. Kim et al. [57] compute the upper and lower memory addresses of the tiles accessed by each computation partition to distribute the arrays. This may lead to unnecessary replication of large portions of the application dataset, also imposing higher data copy costs. For example, $TB_{0,0}$ (i.e., thread block with identifier $0,0$) in Figure 7.2a accesses a 2D tile composed of elements from two different rows of a matrix: $\{(0,0)\ldots(0,3)\} \cup \{(1,0)\ldots(1,3)\}$. However, the linear memory address range defined by the upper and lower bounds of the tile also contains elements that belong to the neighboring tiles in the X dimension of the matrix: $\{(0,0)\ldots(0,15)\} \cup \{(1,0)\ldots(1,3)\}$. Moreover, for output arrays this generates additional data merging operations, as tiles might be wrongly classified as overlapping. For example, the tiles for $TB_{0,0}$ and $TB_{1,0}$ do not overlap, but the analysis in previous works uses their linear address ranges, which do overlap. The per-dimension access pattern information allows AMGE to identify multi-dimensional tiles as non-overlapping entities.
We consider three classes of access patterns: (1) as a function of thread block indices, (2) local to a thread block, and (3) data-dependent. The first class, the most commonly found in GPU applications, is produced when programmers access arrays using affine transformations of the block and thread



(a) BLOCK (b) BLOCK-CYCLIC

Figure 7.2: Computation-to-data mapping examples. TB means thread block.

indices. As a result, threads belonging to thread blocks with contiguous identifiers in one dimension access elements that are contiguous in a dimension of the array. In the `sgemm` example (Listing 7.1), $block_x$ is used to access A_y (line 16) and C_y (line 23), while $block_y$ is used to access B_x (line 13) and C_x (line 23). We use the notation A_i to refer to the ith dimension of array A. This linear relationship allows us to relate thread blocks with the portions of the arrays accessed by them.

The second access pattern type is produced when array dimensions are traversed through loop induction variables or local thread indexes. In the `sgemm` example (Listing 7.1), threads traverse A_x using the induction variables $i + j$ of the nested loops (line 16).

The third type of accesses (i.e., data-dependent) cannot be determined at compile time since the indices are computed with values that are only known at kernel execution time.

Only array dimensions accessed using the first pattern class are eligible for decomposition, since the second class refers to access patterns local to a single thread or thread block, and the third class cannot be determined statically. AMGE uses a novel compiler analysis, for the dimensions that are eligible for decomposition, that determines the thread block-to-data mappings, and classifies them into the most common distribution types: `BLOCK`, `CYCLIC` and `BLOCK-CYCLIC` [47, 59, 28], (illustrated in Figure 7.2 for a 2D array). This analysis tries to map the index expression used in each dimension of each array reference to the following canonical form:

$$m \times t \times G + t \times B + k \tag{7.1}$$

where:

- m: index of the non-contiguous array tile accessed by a thread block (an induction variable or a constant).

- t: array tile size. It is a multiple of the thread block size when threads access different elements or 1 if all threads access the same element of the array's dimension (e.g., the same row in a matrix). The actual thread block size is extracted from the kernel launch parameters at run-time.

- G: number of thread blocks in a dimension of the computation grid (e.g., $grid_x$).

- B: thread block index (e.g., $block_x$). Expressions not using this term do not belong to the first access pattern class.

- k: thread index or a constant. This value does not determine the access patterns across thread blocks.

We find that most GPU array-based computations do actually use this form to distribute the array across thread blocks.
The most common and simple index expression is found when each thread block accesses a single contiguous array tile (i.e., $m = 0$). This expression is classified as `BLOCK`. Another common index expression assigns non-contiguous array tiles to each thread block (i.e., $m > 0$) using a grid-sized stride. This expression is classified as `BLOCK-CYCLIC`. `CYCLIC` is a special case of `BLOCK-CYCLIC`, in which $t = 1$.

Information generation for the runtime

The compiler analysis generates a record with the following information for each dimension of each array used in each GPU kernel: (1) the dimension of the computation grid whose thread block index is used to index the array dimension; (2) the access type (`Read`/`Write`); (3) the distribution type (`BLOCK`, `CYCLIC`, `BLOCK-CYCLIC`). This information is built by combining the values from the different references to the array in the kernel: (1) the intersection of the used computation grid dimensions; (2) the union of the access types; (3) the intersection of the distribution types. Thus, if an array dimension is indexed using block indices of different dimensions of the computation grid or different distribution types, an empty record is generated for that dimension. Array dimensions that are not indexed using the first data access pattern class, get empty records, too. Only array dimensions with non-empty records can be distributed.

7.2.2 Run-time distribution

On each kernel execution, the runtime identifies all possible computation grid decompositions. Then, it uses the compiler-provided information to compute the distribution configurations for each potential computation grid decomposition. Since the computation grid is limited to three dimensions, there are, at most, eight potential decompositions. Finally, the runtime ranks each distribution configuration using an analytical model (details in Section 7.3.3) and distributes the arrays and computation using the highest-ranked configuration. This process is summarized in Algorithm 1.

Algorithm 1 Runtime distribution configuration selection

procedure ChooseDistribution(A, E, g, P)
A: arrays
E: decomposition evaluation function
g: computation grid dimensionality
P: array access pattern from compiler
Returns: computation and array distribution
  $dims \leftarrow$ PartDimCombinations(g)
  $ret_c \leftarrow 0$
  $ret_a \leftarrow \emptyset$
  $score \leftarrow -1$
  for $conf \in dims$ **do**
    $arrays_{conf}, score_{conf} \leftarrow$ E($conf$, A, P)
    if $score_{conf} > score$ **then**
      $score \leftarrow score_{conf}$
      $ret_c \leftarrow conf$
      $ret_a \leftarrow arrays_{conf}$
    end if
  end for
  return ret_c, ret_a
end procedure

Computation grid distribution

For a specific computation grid decomposition, the AMGE runtime defines a grid of GPUs ($gpu_0 \times gpu_1 \times \ldots gpu_{N-1}$) with as many dimensions as decomposed dimensions in the computation grid. Then, the computation grid is decomposed into as many partitions as the number of GPUs in each dimension of the GPU grid, and each partition is assigned to a GPU. We refer to the mapping of the computation grid to the GPU grid as computation distribution configuration.

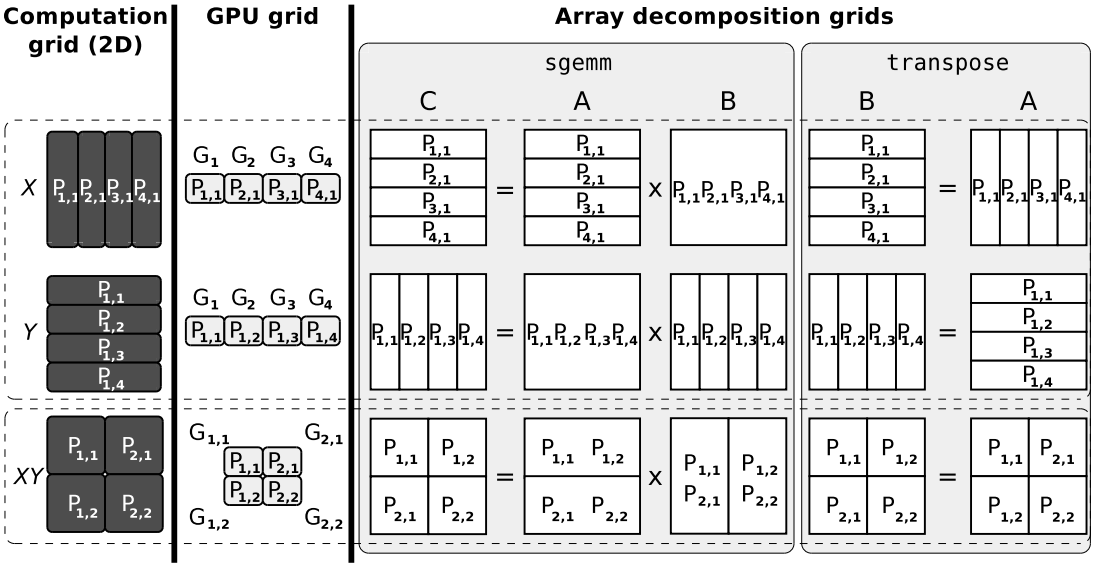


Figure 7.3: Data and computation distribution configurations for `sgemm` and `transpose` on a 4-GPU system. A, B, C are the arrays used in the kernels, $P_{i,j}$ is a partition of the computation grid, $G_{i,j}$ is a GPU in the GPU grid.

Array distribution

The runtime system uses the compiler-generated access pattern information to determine how arrays must be decomposed for a specific computation distribution configuration. An array is decomposed along those dimensions that are indexed with the thread block indices of the decomposed computation grid dimensions. This creates an m-dimensional grid of tiles, where m is the number of decomposed dimensions in the array. Output arrays that are not accessed with any of the thread block indexes of the decomposed computation grid dimensions, are arbitrarily decomposed along their highest-order dimension.

The size and number of tiles depend on the computation distribution configuration. If $block_i$ is used to index L_j, and $grid_i$ is mapped on the k dimension of the GPU grid, then L_j is decomposed into as many tiles as GPUs in gpu_k and distributed among them. Figure 7.3 shows the relationship between the computation grid, the GPU grid and the array decomposition grid in different computation distribution configurations for `sgemm` and `transpose`. In the `sgemm` example, C_x, C_y, A_y and B_x can be potentially decomposed, as they are indexed with thread block indices. For a configuration that decomposes the computation grid on its Y dimension ($grid_y$, second row of Figure 7.3), C_x and B_x are decomposed. However, since $block_x$ is used to index A_y but $grid_x$ is not decomposed, A_y is not decomposed either. In `transpose`, both $block_x$

and $block_y$ are used to index the two dimensions of A and B matrices and are always decomposed. The distribution of tiles among the GPU memories follows the mapping of the computation grid partitions on the GPU grid. For example, in the XY computation grid decomposition of `sgemm` (third row of Figure 7.3), neighboring tiles in C_x are distributed across neighboring GPUs in gpu_y.
Input arrays that are not decomposed are replicated in all GPU memories. Moreover, tiles from input arrays that are not indexed using the thread block indexes of all the decomposed dimensions of the computation grid, are *partially* replicated. In this case, the array tiles are only replicated in the memories of those GPUs that belong to the GPU grid dimensions on which the unused computation grid dimensions are distributed. For example, in `sgemm`, C is indexed with both $block_x$ and $block_y$ and its tiles are not replicated for any computation grid decomposition. However, A is indexed with $block_x$, and B is indexed with $block_y$, only. Thus, A and B are fully replicated in all GPUs for Y and X computation grid decompositions, respectively. In the XY configuration, the tiles in A are distributed across the GPUs in gpu_x and each tile is replicated in all GPUs in gpu_y. For example, the upper tile of A is accessed by both $P_{1,1}$ and $P_{1,2}$ computation partitions, and it is replicated in the memories of GPUs $G_{1,1}$ and $G_{1,2}$. Conversely, the tiles in B are distributed across the GPUs in gpu_y and replicated in the GPUs in gpu_x.

7.3 AMGE implementation details

7.3.1 Array data type

We utilize the UVAS support to place different parts of the array in different GPU memories while having a continuous representation of the array in the virtual address space. Hence, decomposed arrays can be referenced by using regular linearization operations on the indexes: $(a_1, \cdots, a_n) \rightarrow \sum_{i=1}^{n} a_i \times \prod_{j=i+1}^{n} D_j$ where a_i is the index and D_i the number of elements in the ith dimension of the array. Indexes are ordered from the highest-order to the lowest-order dimension. We refer to this scheme as **VM** implementation.
This implementation decomposes arrays with page-size granularity which, in current GPUs, is in the order of a few kilobytes. Nevertheless, we have experimentally determined that current versions of CUDA impose a 1 MB (instead of page-size) granularity to allocate contiguous virtual memory ranges on different GPUs. This produces data distribution imbalance if partitions cannot be stored in balanced-sized groups of 1 MB chunks, resulting in an increased number of remote memory accesses. One solution to reduce the imbalance is

to add padding to the lower order dimensions that are not decomposed. However, achieving perfect balancing using 1 MB chunks can impose a footprint overhead in the order of hundreds of times, especially for arrays' lowest-order dimensions, whose elements are stored contiguously in memory. Another solution is to permute the dimensions of the array to ensure that decomposed dimensions are not contiguous in memory. Unfortunately, this can break memory coalescing [85].
Therefore, we also provide an alternate implementation proposed in [20] that **reshape**s the arrays. In this implementation, each GPU contains a memory allocation that holds all the elements in a partition, and it is padded to the 1MB boundary. The array is reorganized by adding a new dimension for each decomposed one (i.e., strip-mining), that indicates the GPU in which each partition is stored. In each array reference, the original indexes are transformed into a new set of indexes. This approach allows arrays to be decomposed on any dimension as they do not have to be stored contiguously in the virtual address space. However, this flexibility comes at the cost of extra computation. For example, if a 3D volume is decomposed along its highest-order dimension using a `BLOCK` distribution, the index for this dimension a_1 is transformed as follows:

$$(a_1, a_2, a_3) \rightarrow (\left\lfloor \frac{a_1}{D'_1} \right\rfloor, a_1 \bmod D'_1, a_2, a_3) \tag{7.2}$$

where $D'_1 = \left\lceil \frac{D_1}{P_1} \right\rceil$ and P_1 is the number of tiles in the first dimension of the array's decomposition grid. The operations needed to compute the location of the element a_1, a_2, a_3, are divided into: the computations of the offset of the block in the dimension being decomposed, and the linearization of the index within the block:

$$block(a_1, a_2, a_3) = \left\lfloor \frac{a_1}{D'_1} \right\rfloor \times O_1$$
$$linear(a_1, a_2, a_3) = (a_1 \bmod D'_1) \times D_2 \times D_3 + a_2 \times D_3 + a_3$$

where O_1 is the offset between tiles in the first dimension of the array's decomposition grid. Therefore, an extra division and modulo operations are performed in each access to the array, compared with the regular index linearization. For `CYCLIC` and `BLOCK-CYCLIC`, similar transformations are performed.
Providing a generic indexing routine that supports all possible array decompositions can impose an unacceptable performance overhead due to the extra operations needed to transform all the indexes. In order to ensure maximum

performance, our prototype provides different implementations of the indexing routines optimized for the different array decompositions and distribution types.

7.3.2 Source-to-source transformations

Kernel versioning Using specialized indexing routines for each decomposition requires changes in the kernels, as (1) the kernel code must explicitly call the proper versions of the routines, and (2) the array decomposition to be used is not known at compile time. Thus, AMGE generates different kernel versions for all the possible array decompositions of the arrays used in the kernel. In each version, array references use the indexing routines that are optimized for a chosen decomposition. On each kernel execution, when the runtime system selects the distribution for all the arrays, it uses the specialized kernel version for that configuration.

Global thread block identifiers Since computation partitions are executed independently, CUDA assigns new thread block identifiers in the kernel invocations on each GPU. In order to retain the original identifiers, we store the offsets of each computation partition in the memory of each GPU. The compiler modifies replaces `blockIdx` with code that computes the original indexes using these offsets.

Memory consistency model Distributed kernels must honor the memory consistency of the GPU programming model. Data propagation across thread blocks in the GPU model is only guaranteed for atomic memory operations and memory fences. For atomic operations we exploit the hardware support provided by modern system architectures (e.g., atomic operations in PCIe 3.0). GPU-wide memory fences are translated to system-wide memory fences to ensure correctness. Finally, since a GPU may cache remote array partitions, GPU caches are flushed at kernel exit boundary.

AMGE toolchain AMGE provides a toolchain based on the LLVM framework. The first pass performs the array access pattern analysis introduced in Section 7.2.1 on the LLVM IR generated by the CUDA compiler, and generates code to pass the information to the runtime system. The second pass handles `blockIdx` and memory fence translations. The third pass generates specialized kernel versions for the possible array decompositions, and code for the runtime system to retrieve the version for a chosen configuration. Generated code is compiled into the program executable.

7.3.3 Run-time distribution selection policy

As explained in Section 7.2.2, on a kernel call, the runtime system selects the best distribution configuration. Our prototype implements a policy (shown in Algorithm 2) that (1) favors the array implementation that imposes the least overhead, (2) minimizes the number of remote accesses. Thus, the VM implementation is preferred (since it does not impose any indexing overhead) unless it introduces *too many* remote memory accesses due to data distribution imbalance. Our policy ranks the array decompostions given by the runtime system for both VM and reshape implementations. For VM, the score is calculated using the data distribution imbalance introduced by the coarse allocation granularity. The imbalance is computed analytically by counting the bytes that belong to an array partition that should be stored in a different GPU, with respect to the total array size. The cost of this computation is negligible compared to the kernel execution time. When the imbalance exceeds a threshold (in Algorithm 2 the threshold is set to 5%), the score becomes zero (effectively choosing the reshape implementation). If several configurations get the same score, decompositions on the highest-order dimension are preferred because they allow for more efficient CPU↔GPU transfers.

Algorithm 2 Implemented array distribution policy

```
procedure ArrayDistributionPolicy(C, A, P)
C: computation decomposition configuration
A: arrays
P: array access pattern from compiler
Returns: suggested array decomposition and its score
    arrays_info ← DecompositionInfo(A, C, P)
    T ← 0.05                                   ▷ Memory imbalance threshold
    R ← empty
    score ← 1.0
    for array_info ∈ arrays_info do
        i ← ComputeImbalance(array_info, C)
        if i > T then                          ▷ Use reshape implementation
            array ← ArrayReshape(array_info)
            score ← score × (1 − T)
        else                                   ▷ Use VM implementation
            array ← ArrayVM(array_info)
            score ← score × (1 − i)
        end if
        R ← Insert(R, array)                   ▷ Add array decomposition to the set
    end for
    return R, score
end procedure
```

Kernel	**Suite**	**Inputset**	**#Reg Orig**	**#Reg VM**	**#Reg Resh B/C**	**#Reg Resh B-C**	**Array Decompositions**	**Array Distribution**
convolution2D	Parboil 2	A,B: 16K×16K C: 9×9	17	19	19 19 19	22 22 22	X→A,B(∗,BLOCK) C(∗,∗) Y→A,B(BLOCK,∗) C(∗,∗) XY→A,B(BLOCK,BLOCK) C(∗,∗)	A,B:{D_x} C:{R_x} A,B:{D_x} C:{R_x} A,B:{$D_{x,y}$} C:{$R_{x,y}$}
fft1D	Parboil	A,B: 256M	22	24	24	24	X→A(*) B(BLOCK)	A:{R_x} B:{D_x}
reduction	SDK	A,B: 256M	12	12	11	18	X→A,B(BLOCK)	A,B:{D_x}
saxpy	-	A,B: 256M	8	8	8	17	X→A,B(BLOCK)	A,B:{D_x}
sgemm	Parboil 2	A,B,C: 4K×4K	38	33	41 40 42	41 40 42	X→A,C(*,BLOCK) B(∗,∗) Y→A(∗,∗) B(∗,BLOCK) C(BLOCK,∗) XY→A,B(∗,BLOCK) C(BLOCK,BLOCK)	A,C:{D_x} B:{R_x} A:{R_x} B:{D_x} C:{D_x} A:{D_x,R_y} B:{D_y,R_x} C:{$D_{x,y}$}
sgemv	-	A: 8K×8K - B,C: 8K	11	11	11	16	X→A(BLOCK,∗) B(*) C(BLOCK)	A:{D_x} B:{R_x} C:{D_x}
sort_merge_global sort_merge_shared sort_shared	SDK	A_k,A_v,B_k,B_v: 256M	17 17 17	16 16 18	18 18 19	28 27 28	X→A_k,A_v,B_k,B_v(BLOCK) X→A_k,A_v,B_k,B_v(BLOCK) X→A_k,A_v,B_k,B_v(BLOCK)	A_k,A_v,B_k,B_v:{D_x} A_k,A_v,B_k,B_v:{D_x} A_k,A_v,B_k,B_v:{D_x}
stencil2D	-	A,B: 16K×16K + halos	17	18	17 17 19	23 19 26	X→A,B(∗,BLOCK) Y→A,B(BLOCK,∗) XY→A,B(BLOCK,BLOCK)	A,B:{D_x} A,B:{D_x} A,B:{$D_{x,y}$}
stencil3D	Parboil 2	A,B: 1K×1K×512 + halos	24	26	30 29 33	34 31 41	X→A,B(∗,∗,BLOCK) Y→A,B(∗,BLOCK,∗) XY→A,B(∗,BLOCK,BLOCK)	A,B:{D_x} A,B:{D_x} A,B:{$D_{x,y}$}
transpose	SDK	A,B: 16K×16K	16	14	17 16 18	24 23 26	X→A(∗,BLOCK) B(BLOCK,∗) Y→A(BLOCK,∗) B(∗,BLOCK) XY→A,B(BLOCK,BLOCK)	A,B:{D_x} A,B:{D_x} A:{$D_{x,y}$} B:{$D_{y,x}$}
vecadd	-	A,B,C: 256M	10	10	10	18	X→A,B,C(BLOCK)	A,B,C:{D_x}

Table 7.1: Dense benchmarks used for the evaluation of AMGE.

7.4 Experimental methodology

All experiments were run on a system containing a quad-core Intel i7-3820 at 3.6 GHz with 64 GB of DDR3 RAM memory, and 4 NVIDIA Tesla K40 GPU cards with 12 GB of GDDR5 each, connected through a PCIe 3.0 in x16 mode (containing two PCIe switches like in Figure 6.1b, page 73). The machine runs a GNU/Linux system, with Linux kernel 3.12 and NVIDIA driver 340.24. Benchmarks were compiled using GCC 4.8.3 for CPU code and NVIDIA CUDA compiler 6.5 for GPU code. Execution times were measured using the CUPTI profiling library that provides support for sampling and nanosecond timing accuracy. For runs with more than one GPU, graphs show the time for the slowest GPU.

We evaluate AMGE using a number of dense scientific computations that use different computation and array access patterns. The list of benchmarks is summarized in Table 7.1. Some of them are found in the Parboil benchmark suite [48], some in the NVIDIA SDK, and the rest have been developed in-house. These benchmarks are selected to provide a good variety of access patterns and thus challenges. Both CPU and GPU codes have been modified to use the `ndarray` data type instead of the flat 1D arrays which are commonly used. The benchmarks have been compiled using our toolchain and linked with our runtime system. The `ndarray` implementation has an impact on the register usage count of the kernels (columns 4-7). In the column titles, "Orig" stands for original, "VM" for virtual memory and "Resh" for reshape, while "B" stands for `BLOCK`, "C" for `CYCLIC` and "B-C" for `BLOCK-CYCLIC`. `BLOCK` and `CYCLIC` are in the same column ("#Reg Resh B/C") as they use the same number of registers.

Columns 8 and 9 show the array decompositions and distributions suggested by AMGE for the possible computation distribution configurations. A, B, C are the names of the arrays used in the computation. In the case of the kernels in merge sort, a suffix has been added for keys (A_k, B_k) and values (A_v, B_v). In the last column, "D" stands for distribution and "R" for replication in the GPU grid. X and Y make reference to the decomposed dimensions of the computation grid. Array decompositions are shown using the notation in HPF [59], but dimensions are ordered (left to right) from highest to lowest order as they are stored in memory. For example, the `sgemv` configuration says that for a computation decomposition on X, the A matrix is decomposed on its Y dimension and the C vector is decomposed on its X dimension. Tiles in A and C are distributed across the GPUs in the X dimension of the GPU grid while B vector is replicated.

We also analyze two sparse computations in order to prove that AMGE is able to execute irregular workloads: SpMV (i.e., sparse matrix vector

Kernel	Suite	Decomposition	Distribution
SpMV - CSR	CUSP	A_row_offsets(BLOCK)	D_x
		A_col_indices(∗)	R_x
		A_values(∗)	R_x
SpMV - COO	CUSP	A_rows(∗)	R_x
		A_cols(∗)	R_x
		A_values(∗)	R_x
SpMV - ELL	CUSP	A_col_indices_trans(∗,∗)	R_x, y
		A_values_trans(∗,BLOCK)	D_y,R_x
SpMV - HYB	CUSP	*COO + ELL*	
SpMV - JDS	CUSP	A_col_indices_trans(∗)	R_x
		A_col_begin_index(∗)	R_x
		A_row_perm(∗)	R_x
		A_nzcnt(BLOCK)	D_x
		A_values_trans(∗)	R_x
BFS	Rodinia	graph_nodes(BLOCK)	D_x
		graph_mask(BLOCK)	D_x
		graph_edges(∗)	R_x
		graph_updating_graph_mask(∗)	R_x
		graph_visited(∗)	R_x
		graph_cost(∗)	R_x

Table 7.2: Sparse benchmarks used for the evaluation of AMGE.

multiplication) and BFS (i.e., breadth first search).
The SpMV computation kernel implementation is highly dependent on the used sparse matrix format. GPU implementations for the DIA, CSR, COO, ELL and HYB formats are proposed in [27]. The DIA (from *diagonal*) format is meant to represent matrices with values in their diagonal. CSR (i.e., Compressed Sparse Row) stores the column indices for each element, and an array of pointers to the beginning of each row. COO (from *coordinate*) stores the row and column indices of every value. Each thread computes one element product, and then a segmented reduction is performed to sum the values computed by different threads. ELL (from *ELLPACK*, a system for solving elliptic boundary value problems) pads all rows to the maximum row size so that accesses to the elements of each row are aligned. Values are also transposed so that adjacent threads can read elements from different rows in a coalesced manner. The ELL format works very well for matrices in which rows have similar sizes. Otherwise, (1) if adjacent rows have very different sizes will produce thread divergence, (2) many values in the rows will be unused and the memory requirements might become unmanageable. The HYB (from *hybrid*) tries to fix this problem by mixing the COO and ELL formats. Thus, ELL can be used to efficiently store the first n elements of each row, and COO is used to store the rest of elements. The Parboil benchmark suite also provide a highly-efficient implementation of SpMV using the JDS (i.e., Jagged Diagonal Storage) format [84]. JDS rearranges rows according to the number of nonzero elements per row. It can be viewed as a ELL

format that minimizes the imbalance among adjacent rows to avoid thread divergence. We have ported the implementations for COO, CSR, EEL, HYB provided in the CUSP library [9], and JDS from Parboil, to AMGE.
Table 7.2 shows the decomposition and distribution configurations determined by the AMGE runtime. All the kernels use a 1D computation grid and, therefore, its partitioned on its X dimension. It is important to note that many arrays are replicated since they are accessed through data-dependent values and it is impossible for the compiler to infer an access pattern. The decomposition and distribution of the input and output vectors is not shown but it is the same for all the implementations: the input vector is replicated in all GPUs (i.e., R_x), and the output vector is partitioned and distributed across memories (i.e., D_x).
The performance in SpMV completely depends on the shape of the sparse matrix. In order to use representative matrices, we use the 10 first matrices in the Matrix Market database [15]. They are shown in Figure 7.4. These matrices are used in different scientific research areas: quantum chromodinamics, finite element analysis, structural engineering, circuit design, and fluid dynamics.
BFS is an algorithm used to traverse trees or graph data structures. It starts in one arbitrary node and visits its neighbors. In the next step, the same process is performed for each of the neighbors in the last step. The BFS implementation in Rodinia [35] uses a two-kernel algorithm that avoids communication across thread blocks. It uses arrays to describe nodes, edges, and keeps track of the nodes that have been already visited. These pair of kernels are called iteratively until there are no remaining nodes to visit. We use the biggest graph inputset shipped in the Rodinia benchmark suite.

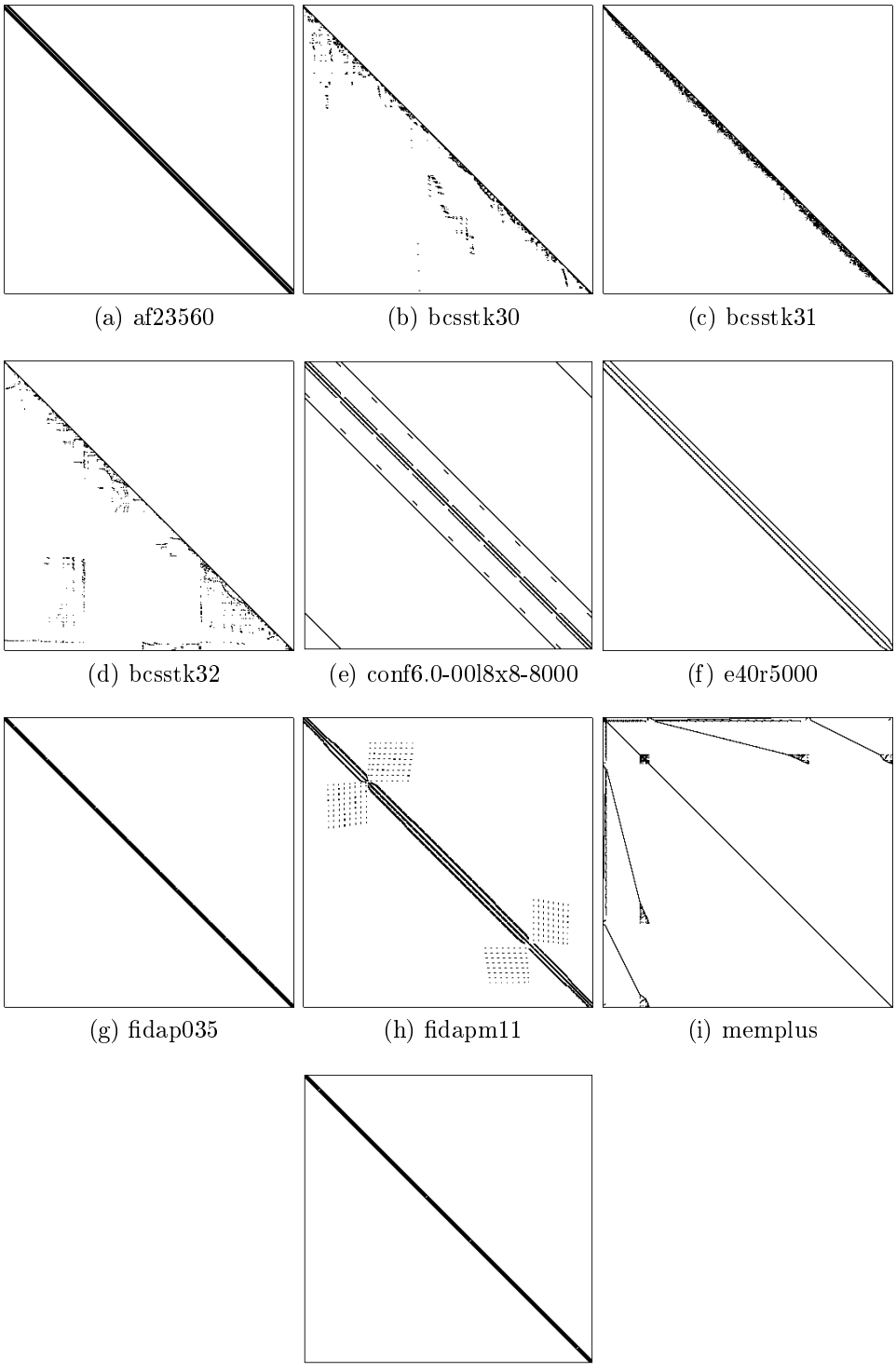
7.5 Performance evaluation

In order to evaluate the efficiency of AMGE we evaluate several aspects, from the overhead introduced by the array data type, to the performance scalability and the costs due to remote memory accesses. We also compare the performance of AMGE with prior works.

7.5.1 Indexing overhead

There are two main costs that are imposed by our array data type:

- Extra operations in the indexing routines. Both *VM* and *reshape* array implementations linearize the indexes in order to determine the location



(a) af23560 (b) bcsstk30 (c) bcsstk31

(d) bcsstk32 (e) conf6.0-00l8x8-8000 (f) e40r5000

(g) fidap035 (h) fidapm11 (i) memplus

(j) s3dkt3m2

Figure 7.4: Sparse matrices used in the SpMV benchmarks.

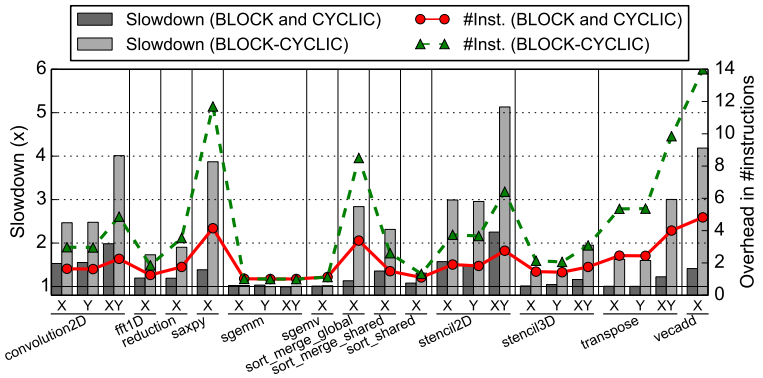


Figure 7.5: Grey bars show the slowdown imposed by the indexing routines for the *reshape* array implementation compared to the baseline (left axis). Lines indicate the increase in number of executed instructions (right axis).

of the referenced element. The *reshape* implementation performs strip-mining, which adds some additional operations. These routines can also increase the register usage count, thus impacting on the occupancy of the GPU cores. This subsection analyzes this potential problem.

- Memory mapping imbalance. This introduces extra remote memory accesses but it is only suffered by the *VM* implementation due to the mapping granularity in CUDA. We analyze this problem in the next subsection.

Table 7.1 shows that register utilization greatly varies depending on the used `ndarray` implementation. The *VM* implementation only performs the index linearization and, therefore, the register count is similar to, or even slight lower than the original version of the benchmarks. *reshape*, on the other hand, uses more registers in most of the kernels, especially in `BLOCK-CYCLIC` decompositions and those configurations in which arrays are decomposed along several dimensions.
Figure 7.5 shows the overheads imposed by the indexing routines for the *reshape* implementation in the dense computation benchmarks running on a single GPU. While AMGE suggests the utilization of the BLOCK distribution type in all kernels, we study the overhead of the routines for all the distribution types. Grey bars show the slowdown imposed by the indexing overhead of the data distributions for each computation distribution configuration (left axis). Lines represent the increase in number of executed instructions due to the extra operations performed on the indexes (right axis). `BLOCK` and `CYCLIC` are grouped (dark gray bar and solid line) as they perform very similar transformations on the indexes and the performance is

virtually the same (±1%). `BLOCK-CYCLIC` (light Gray bar and dashed line) is consistently the slowest implementation (up to 4.48×) and the one that executes more instructions, too. This is caused both by the extra executed instructions and the lower achieved occupancy in the GPU due to the increased number of registers. Slowdowns are large for kernels in which thread blocks perform little work (`convolution2D`, `reduction`, `saxpy`, `sort_merge_*`, `stencil2D`, `transpose` and `vecadd`). Results for *VM* implementation are not shown because performance is within ±5% of the baseline in all kernels.

7.5.2 Multi-GPU performance

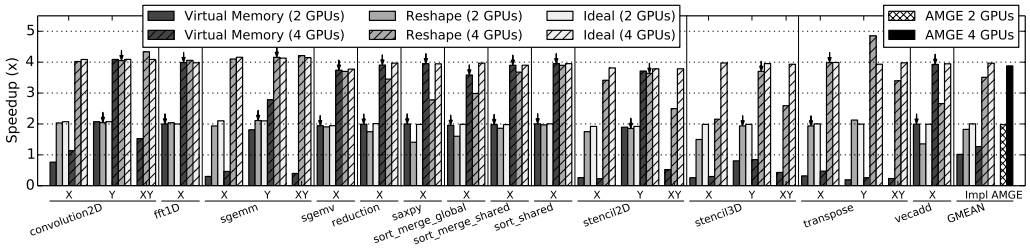


Figure 7.6: Speedup over baseline for different computation decomposition configurations using *reshape* and *VM* implementations. Arrows point to the configuration chosen by the runtime system for each kernel. Results shown for 2/4 GPUs.

Figure 7.6 shows the speedup achieved by AMGE on our multi-GPU system for all possible distribution configurations. Results are shown for the *VM* and *reshape* implementations of the `ndarray` data type, and for an *ideal* implementation with optimal data distribution (no remote accesses). Bars labeled with "Impl" show the geometric mean for the three implementations of the speedups achieved in each kernel by the best computation distribution configuration. The *reshape* implementation exhibits linear speedups (1.91× and 3.54× on average for 2 and 4 GPUs, respectively) for all kinds of distribution configurations in most kernels. The main exceptions are X and XY configurations in `stencil3D` due to remote memory accesses, and `saxpy`, `vecadd`, `sort_merge_global` and the XY configuration in `stencil2D` due to the overhead of the indexing function. The *VM* implementation outperforms *reshape* in some configurations in which the array partitions are large enough not to suffer from imbalance due to the memory allocation granularity imposed by CUDA, but performs very poorly in the other configurations, producing lower speedups on average (1.02× and 1.27× for 2 and 4 GPUs). For example, in `transpose` at least one of the matrices must be decomposed along its X dimension, which is contiguous in memory, thus leading to an

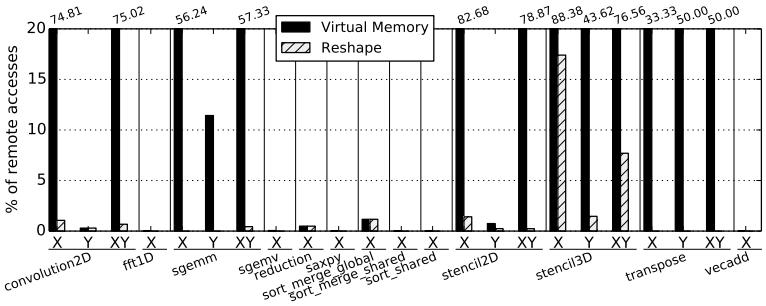


Figure 7.7: Memory requests served by remote GPUs.

imbalanced data distribution. Therefore, performance is poor for *VM* in all configurations in `transpose`. Another example is `stencil3D`, for which the *Y* distribution configuration should provide reasonable performance since each plane of the volume can be distributed across GPUs. Nevertheless, the size of each plane still produces an imbalanced distribution. We study this example in more detail in Section 7.5.3.

On each kernel call, AMGE's runtime system tries to choose the best computation and data distribution configuration. In this evaluation, AMGE implements the policy introduced in Section 7.3.3, using a 5% ditribution imbalance threshold to choose between *VM* and *reshape* implementations. We choose this number because we have determined experimentally (Figure 6.7) that the GPUs in our evaluation system can hide the costs of up to 10% remote accesses depending on (1) how they are distributed in the kernel execution (Batch or Spread); and (2) the computational intensity of the kernel (FLOPS/load). We can see that, using this value, the policy correctly selects the best performing configuration for most of the kernels. Figure 7.6 highlights with an arrow the chosen configurations. The average performance across all the benchmarks (i.e., bars labeled with "AMGE") is 1.98× and 3.89× for 2 and 4 GPUs; very close to *ideal*.

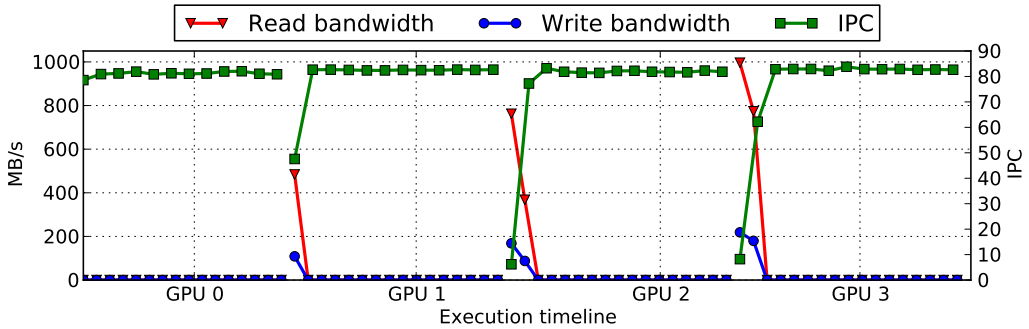
7.5.3 Impact of remote accesses on performance

Remote accesses can happen because either, some region of an array partition is accessed from several GPUs (e.g., in a stencil patterm), or the data distribution imbalance introduced by the *VM* array implementation. We measure the amount of remote memory accesses for all the computation decomposition configurations.

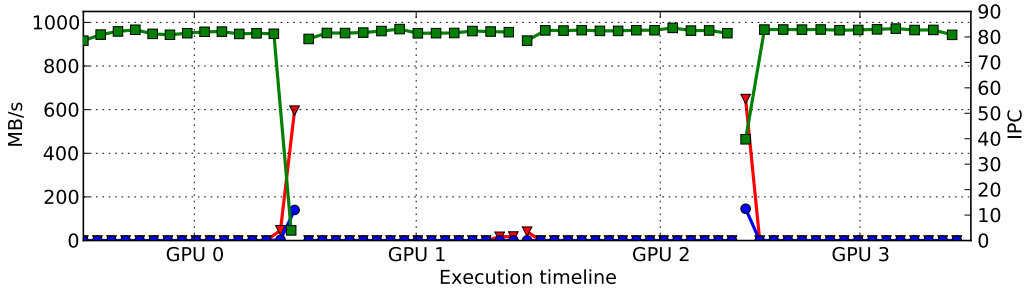
Figure 7.7 shows the percentage of accesses to RAM memory that are served by remote GPUs in all the kernels when they are distributed across 4GPUs. *reshape* eliminates the need for remote accesses in most of the configurations.

Only kernels in which computation partitions share some data use them (e.g., `convolution2D`, `stencil{2,3}D`). The worst cases are the X and XY decompositions for `stencil3D` in which 17.43% and 7.61% of accesses to memory are remote, respectively. This is the reason why these configurations show poor speedups in Figure 7.6. In contrast, *VM* introduces a lot of remote accesses in many configurations due to the memory allocation granularity in CUDA.

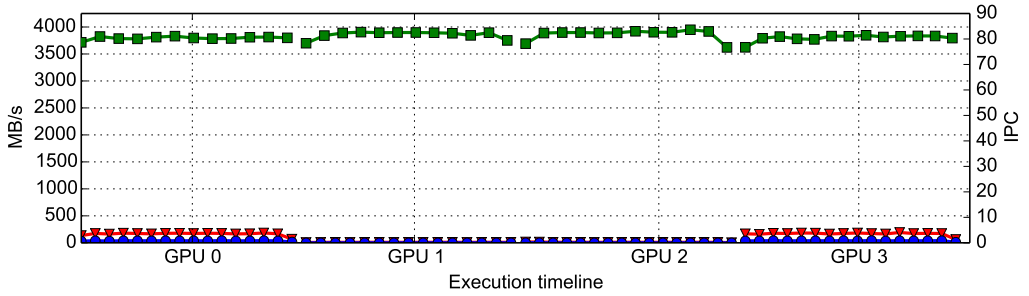
Fighting data distribution imbalance in *VM* The dimensions of the arrays in `stencil2D` and `stencil3D` kernels make it difficult to evenly split them using 1 MB granularity, causing imbalance and, therefore, remote memory accesses. The computation distribution configuration that we study is Y. Using this configuration, the volumes of `stencil3D` are distributed by allocating partitions of each plane alternatively in different GPUs. The size of each plane (1K×1K + halos) produces an imbalanced distribution (2/1/1/1MB for 4GPUs). This results in excessive communication that limits the performance (0.87× and 0.91× for 2 and 4 GPUs). Adding padding to the X dimension of the volume to obtain a balanced distribution results in a 127.01× memory footprint increment. Having a 4 KB granularity would reduce this overhead to 1.49×. Using more friendly problem sizes that do not produce imbalance results in much improved performance, reaching linear speedups of 2.08× and 3.95× for 2 and 4 GPUs.
The larger size (16K×16K + halos) of the plane in `stencil2D` allows for a more balanced distribution of data (65/64/64/64MB for 4GPUs). However, there are still some effects on the performance. Figure 7.8a shows the memory bandwidth consumption due to remote loads and stores (left axis), and the number of instructions per cycle (right axis) during the execution of the `stencil2D` kernel for each GPU in the system. For the sake of clarity, we concatenate the execution timelines of the four GPUs, although they execute in parallel. GPU 0 does not perform any remote memory access and the IPC (instructions per cycle) remains stable during the kernel execution. GPUs 1, 2 and 3 perform remote memory accesses to the previous GPU memories (note that the imbalance increases with the GPU identifier). Remote accesses degrade the performance of the GPU as reflected in the lower IPC. Using padding to obtain a fully balanced array distribution would increase the memory footprint by 7.99×. Instead, we reduce the imbalance by offsetting the beginning of the array so that GPUs 0 and 3, and GPUs 1 and 2 perform the same amount of remote memory accesses (Figure 7.8b). The improved balancing lowers the remote memory bandwidth consumption (2.5 GBps vs 4 GBps), and the period in which remote accesses are performed is shorter.



(a) Imbalanced distribution



(b) Balanced distribution



(c) Balanced distribution + transposed thread block scheduling

Figure 7.8: Execution timeline of `stencil2D` for 4 GPUs.

Benchmark	Conf	Kim[57]	Lee[61]	AMGE
convolution2D	X	38.7K×38.7K	38.7K×38.7K	77.4K×77.4K
	Y	38.7K×38.7K	38.7K×38.7K	77.4K×77.4K
	XY	38.7K×38.7K	38.7K×38.7K	77.4K×77.4K
fft1D	X	750M	750M	3G
reduction	X	2.99G	2.99G	11.99G
saxpy	X	6G	6G	6G
sgemm	X	31.6K×31.6K	31.6K×31.6K	44.7K×44.7K
	Y	44.7K×44.7K	31.6K×31.6K	44.7K×44.7K
	XY	38.7K×38.7K	31.6K×31.6K	48.9K×48.9K
sgemv	X	77.4K×77.4K	38.7K×38.7K	77.4×77.4
sort	X	750M	750M	3G
stencil2D	X	38.7K×38.7K	38.7K×38.7K	77.4K×77.4K
	Y	48.9K×48.9K	38.7K×38.7K	77.4K×77.4K
	XY	44.7K×44.7K	38.7K×38.7K	77.4K×77.4K
stencil3D	X	$1.1K^3$	$1.1K^3$	$1.8K^3$
	Y	$1.3K^3$	$1.1K^3$	$1.8K^3$
	XY	$1.2K^3$	$1.1K^3$	$1.8K^3$
transpose	X	48.9K×48.9K	38.7K×38.7K	77.4K×77.4K
	X	48.9K×48.9K	38.7K×38.7K	77.4K×77.4K
	XY	54.7K×54.7K	38.7K×38.7K	77.4K×77.4K
vecadd	X	4G	4G	4G

Table 7.3: Maximum problem size for a 4-GPU system in AMGE and in the related work.

This results in a 8.2% execution speedup on the slowest GPU.

Reducing instantaneous bandwidth demands In `stencil2D`, memory accesses concentrate at the beginning/end of the kernel execution because the default thread block scheduler in the GPU issues thread blocks that are contiguous in the X dimension in order. Thus, thread blocks that access the boundaries of the matrix partitions tend to execute concurrently, increasing the instantaneous bandwidth demands and reducing the achieved IPC. We evaluate the performance of a thread block scheduler that issues thread blocks that are contiguous in the Y dimension instead. We emulate it by transposing the mapping of thread block identifiers on the matrices. Figure 7.8c shows that, using this scheduler, remote accesses are distributed throughout all kernel execution, thus reducing the instantaneous bandwidth demands to 200MBps. Now the IPC is not affected, since the cost of remote accesses is hidden with the execution of other thread blocks that only perform local accesses, reducing execution time by a 5.8%.
These results confirm the conclusions from previous chapters, that argues that providing mechanisms to control thread block scheduling can help reducing the overhead of remote memory accesses.

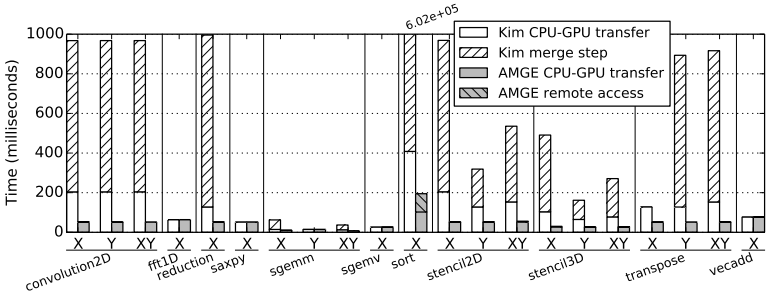


Figure 7.9: Overhead of the coherence mechanisms in AMGE and in the related work [57].

7.5.4 Comparison with previous works

We measure the benefits of AMGE over the solutions proposed by Kim et al. [57] and Lee et al. [61]. Since the source code of previous works is not publicly available, we approximate the comparison by measuring, for each solution, the maximum problem size, the memory coherence overhead, and the kernel execution performance.

Memory footprint overhead Thanks to the array dimensionality information, AMGE performs much more space-efficient data decompositions than previous works. We compute the maximum problem size that can be executed by the three solutions on our 4-GPU system, using the information in [57, 61]. Table 7.3 shows that AMGE is able to run bigger problem sizes than previous works for most benchmarks, which is one of the major motivations for using multiple GPUs, especially on those that work on multi-dimensional arrays.

Coherence overhead AMGE ensures coherence by not replicating output arrays and using remote memory accesses when needed. The related work relies on replication and a merge step after kernel execution. Besides, data needs to be copied from CPU to GPU memories before kernel execution. Figure 7.9 shows the overhead of both solutions. AMGE shows lower CPU/GPU transfer times in most cases (CPU-GPU transfer), because of the more efficient partitioning of multi-dimensional data structures, which requires smaller regions of the arrays to be resident in each GPU memory. The overhead of the merge step is even larger compared with the cost of remote accesses. The most extreme case is the `sort` benchmark, since it executes kernels iteratively and the merge step needs to be performed after each kernel call.

Kernel performance Previous works replicate arrays to ensure that all accesses are local. We approximate the kernel performance that would have been achieved by prior works by looking at the *ideal* bars (i.e., no remote accesses) in Figure 7.6. We see that the implementation/configuration chosen by the AMGE runtime matches the performance of the *ideal* implementation in most of the benchmarks.

Summary AMGE provides similar kernel performance than previous works, while having much lower coherence overhead and being able to handle much larger problem sizes.

7.5.5 Sparse benchmarks

In order to measure the performance of AMGE for irregular workloads, we run the SpMV and BFS benchmarks.
Figure 7.10 shows the execution time of the SpMV benchmarks for 1, 2 and 4 GPUs. We can see that COO is by far the worst-performing sparse matrix format. ELL cannot be used for the `bcsstk30`, `bcsstk31`, `bcsstk32`, and `memplus` matrices because the row size is not balanced and the memory requirements are too high. JDS is the best-performing format for most of the matrices.
Figure 7.10 also compares the different array implementations. We can see that using the VM implementation (Figure 7.10b), only the JDS matrix format shows performance improvements when increasing the number the GPUs. This is mainly because of the large memory mapping granularity, that introduces too many remote accesses. The reshape implementation (Figure 7.10c) exhibits better performance for the CSR and ELL implementations. COO and HYB formats (that also uses COO for a region of the matrix), on the other hand, are slower. The performance achieved by the reshape implementation is very close to the performance that would be achieved if there were no remote accesses (Figure 7.10a).
One of the main challenges that present irregular workloads is that it is difficult to automatically decompose computation in such a way that each GPU performs the same amount of work. For example, in SpMV, most implementations create a thread or warp per output element but the length of the rows can be very different. Figure 7.11 shows the computation imbalance for each of the matrices when being executed on 4 GPUs. The imbalance is computed as the relative execution time of the slowest GPU compared to the fastest one. We can see that JDS is the one that introduces more imbalance. This is because JDS stores rows sorted by their length. Thus, threads in the thread blocks with the lower identifiers will compute the rows

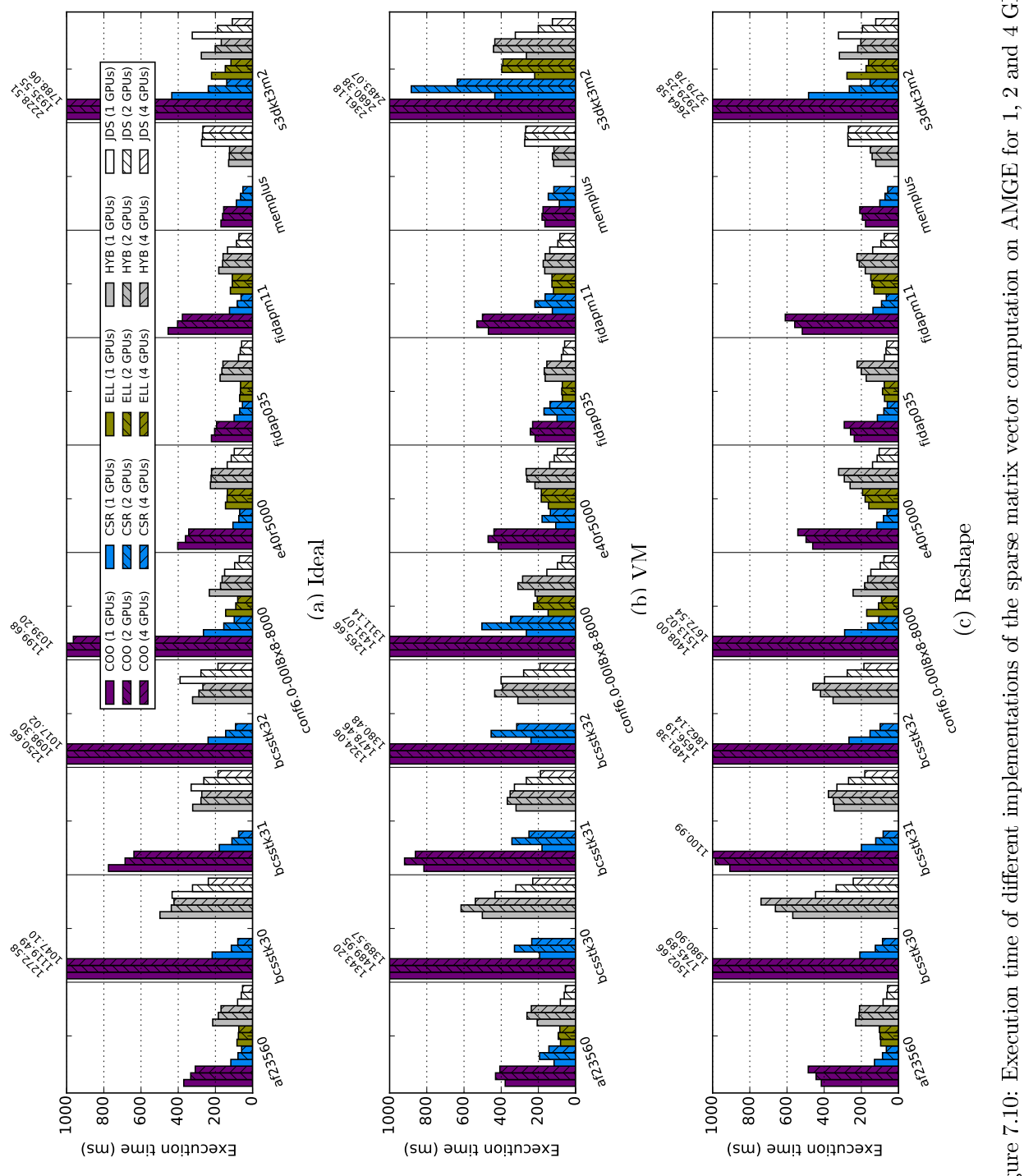


Figure 7.10: Execution time of different implementations of the sparse matrix vector computation on AMGE for 1, 2 and 4 GPUs.

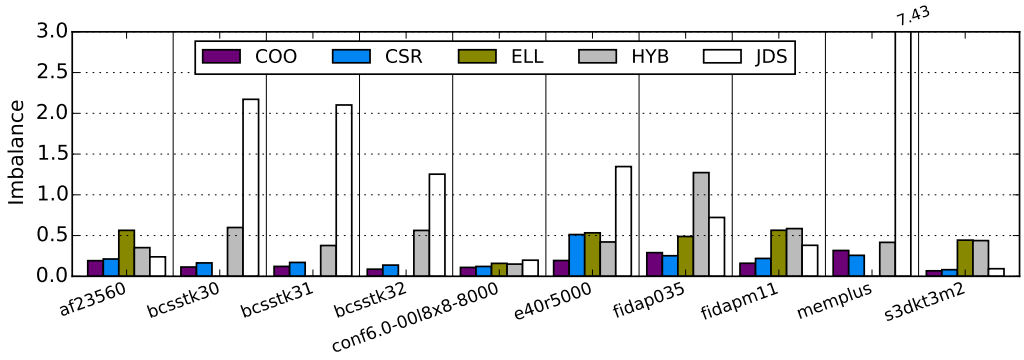


Figure 7.11: Computation imbalance in the SpMV benchmark (4 GPUs).

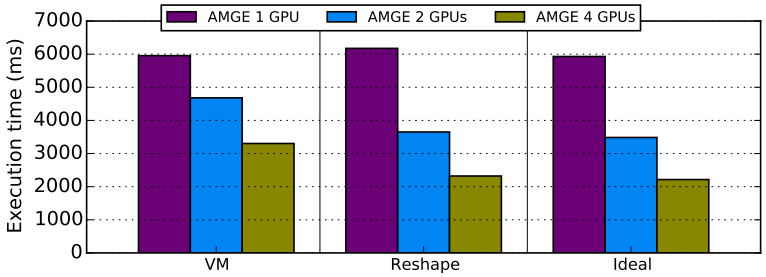


Figure 7.12: Performance scalability in the BFS benchmark.

with the highest number of elements. Since AMGE decomposes the computation grid uniformly, the first GPU will execute the thread blocks that access the longest rows, thus producing imbalance. Without this problem, JDS would yield much better scalability results (e.g., see the `memplus` matrix in Figure 7.10). ELL being a similar format to JDS, does not suffer from this problems because rows of all sizes are distributed to all GPUs. This limitation is intrinsic to the JDS sparse matrix format and not related to AMGE. We believe that AMGE-aware libraries could provide with additional information to the AMGE runtime, so that it can perform a more balanced work distribution in these types of scenarios.

Performance results for the BFS benchmark are shown in Figure 7.12. We can see that AMGE is able to scale the performance when using several GPUs. The reshape implementation shows slightly worse results than VM for 1 GPU due to the higher array indexing costs. However, reshape is better for higher GPU counts because, like in previous experiments, VM introduces more remote memory accesses. The good performance in BFS is due to the low computation imbalance (0.31) introduced by the computation distribution.

Parallelizing irregular workloads is an open problem and it is challenging even for hand-tuned implemenation. While AMGE does not provide linear speedups in all cases like with dense benchmarks, it exhibits speedups in most of the cases, especially for kernels that do not suffer from large computation distribution imbalance across thread blocks.

7.6 Summary

Modern GPUs provide mechanisms that enable efficient auto-parallelization. In this chapter we introduce AMGE, a programming interface, compiler support and runtime system that enables multi-GPU execution of computations written for a single GPU. Thanks to remote memory accesses, AMGE imposes much lower memory footprint and coherence management overheads than previous works. We also demonstrate that transparent data distribution can be efficiently implemented on current GPUs using the UVAS and compiler/runtime-assisted code versioning. Using the array data type provided in AMGE results in shorter and cleaner code, too. AMGE achieves almost linear speedups for most of the dense benchmarks on a real 4-GPU system with an interconnect with moderate bandwidth. Further performance improvements can be achieved by reducing the virtual memory mapping granularity exposed by CUDA and by allowing programmers to tune the thread block scheduling policy. Irregular computations can also benefit from AMGE. However, we believe that AMGE-aware libraries are needed in order to solve the inherent limitations of transparent work distribution for irregular computations.
We believe that AMGE could be used in future systems such as NVIDIA Pascal boards to automatically scale the performance of GPU kernels to multiple GPUs.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Graphics Processing Units provide computation capabilities and energy efficiency that is unmatched by regular CPUs for many types of programs. Many High Performance Computing systems include several GPUs in each node in order to increase their computational power. Further, GPU vendors are announcing products that pack several GPUs in a single board. Thus, programs need to fully utilize all the GPUs in the system ir order to provide good execution efficiency.
GPUs are becoming fully-featured processors with similar capabilities to general purpose processors, such as virtual memory and inter-GPU communication. However, programming models do not exploit these hardware characteristics to facilitate the task of writing programs for multiple GPUs. CUDA and OpenCL still show GPUs as separate devices, and programmers need to explicitly manage the GPU memories and data communication.
In this thesis we have shown that the higher-level abstractions that HPE provides to programmers not only simplifies application source code, but also can yield better performance. Removing the 1:1 host-thread to GPU mapping avoids unnecessary inter-thread synchronization. Direct GPU-to-GPU transfers also allows for transparent highly-tuned copies (i.e., double buffering, PCIe full-duplex exploitation) performed by the runtime. The UVAS also enables P2P communication between GPUs without host intervention (thus boosting memory exchange bandwidth), as virtual addresses correspond to a single physical location in the system. The extension of the ADSM to multi-GPU systems allows memory objects to be declared once and

used both in CPU and GPU codes in multi-GPU applications. Multithread-ADSM also enables performing prefetching behind the scenes, which can give better performance that hand-tuned memory transfers. We have provided a production-level implementation of Reverse Time Migration on top of HPE. This dissertation has also shown that multi-GPU systems can be programmed like shared memory machines. The resulting code for computations with data dependencies between GPUs is much simpler, as they access data remotely and, therefore, do not need to perform explicit a data exchange between GPU memories. We have analyzed the performance characteristics of current interconnects and of the remote access mechanism. Current GPUs are able to hide the costs of a 10% of remote memory accesses if the kernels produce high GPU core occupancy and the accesses are spread through the whole kernel execution. Future interconnects such as NVLink will provide much higher bandwidths and lower latency that will allow to hide the costs of a higher number of remote accesses. We have also measured the impact of GPU hardware features such as caching of remote memory accesses.
In the final part of the thesis we have proposed AMGE, a programming framework for shared memory multi-GPU systems that automatically decomposes computation and data and distributes them across all the GPUs in the system. AMGE further reduces the size of the host code because it is written as if there was a single GPU in the system. We have shown that current data-parallel programming models for GPUs such as CUDA allow for transparent distribution of computation across GPUs. Data decomposition and distribution is automatically determined by compiler analisys that discovers the access patterns on multi-dimensional arrays, and a runtime system that also takes into account the shape of the computation grid. AMGE automatic data distribution avoids bad decisions made by programmers. Performance results show linear speedups for a wide range of dense computations. Although they might suffer from lower scalability if work is not balanced between thread blocks, sparse computations can also benefit from AMGE.

8.1.1 Impact

Several of the works presented in this thesis have been included in commercial products.

- RTM: the production-grade GPU implementation of RTM used in Repsol is based on our work.

- CUDA: since CUDA 4, NVIDIA is adding most of the features of the HPE model.

- AMD: ships a limited version of the HPE model for OpenCL that has been developed in collaboration with Multicoreware Inc. in their AMD APP SDK.

8.2 Future work

The work presented in this dissertation opens new research lines for the programmability of multi-GPU systems. We expect to pursue future work in the following fields.

8.2.1 Full system memory coherence

Current implementation in CUDA 6.0 of the multithreaded ADSM memory model (i.e., Unified Virtual Memory or UVM) is asymmetric: the CPU is the only processor that performs coherence operations. When a kernel is launched, the CPU ensures that all shared data is copied to the GPU memory. Therefore, data cannot be concurrently accessed from CPU/GPU (even different regions of the same allocation), which leads to the underutilization of CPUs or GPUs in some cases.
We will explore the utilization of more advanced memory protection mechanisms on GPUs. For example, support for paging in the GPU will enable lazy migration of data to the GPU memory. This will avoid transferring data which is not accessed by the GPU kernel. Moreover, coherence operations could be executed from the kernel code to implement more advanced protocols. Newly proposed mechanisms such as thread block preemption [88] could also be used to save the context of the thread block that triggers the interrupt and switching to a different thread block while the page fault is being serviced.
We also will look into pre-fetching policies to further improve performance. Migrating one page at a time will introduce a lot of overhead (due to the costs of triggering an interrupt and executing the page fault handling routine) and will provide very low data transfer efficiency (PCIe favors large data transfers). However, due to the highly parallel nature of GPUs the set of pages accessed at any time is potentially very large and, therefore, we believe that traditional pre-fetching policies will not work, and novel policies will need to be proposed.

8.2.2 Hiding the costs of remote accesses

We have shown that thread block scheduling policies are key to distribute remote accesses through the whole kernel execution. Unfortunately, current GPUs implement a single fixed scheduling policy. Two approaches can be taken:

- Implement a hardware-only policy that takes into account very-long-latency operations and mixes thread blocks that use them with other thread blocks.
- Include some hooks in the hardware that can be controlled by system/programmers. A first example can be the grid dimension for which thread blocks are scheduled in order.

The GPU memory hierarchy is not optimized to handle remote memory accesses. Currently remote accesses can only be cached in the R/O L1 cache in each GPU core. Adding a shared L2 R/O cache would allow to reuse data that has been previously cached by a different core, further minimizing the amount of requests to remote GPU memories. Another improvement could be adding remote data prefetching to perform larger requests that better utilize the PCIe interconnect.

8.2.3 Memory placement

AMGE provides automatic data placement for array-based data structures. While this has proven to be very effective for dense computations, irregular workloads might assign more work to some thread blocks. Since this work distribution is dynamic and cannot be determined at compile time, we propose providing AMGE-aware data structures that use user-provided metrics to determine how computation and data should be distributed across GPUs. This could be transparent to programmers for libraries that solve specific problems or with annotations provided by programmers (e.g., weighting functions).
Currently, AMGE uses the best data placement for each kernel. This can produce data reshape if the decomposition or distribution of an array is different for different kernels. We can speedup this reshaping by implementing efficient in-place array reshaping (this translates to a transpositon) like the ones proposed in [32, 87]. Another option is to perform a compile-time estimation to compare the costs of reshaping with the costs of performing remote accesses.
Finally, thanks to GPU paging we could propose dynamic migration and replication that adapts to any type of workload at run time.

8.2.4 Extensions to other environments

This thesis focuses on High Performance Computing environments and workloads, because they currently are the most important use cases for multi-GPU execution. However, new types of systems and applications will benefit from our proposals. For example, cloud computing is used to dynamically adapt the computing resources to the demands of the workload. However, clusters for cloud computing are typically heterogeneous as they contain nodes with different capabilities and device topologies. HPE offers an effective abstraction to virtualize GPU devices that enables *GPU as a service* in the cloud. On the other hand, such systems have different requirements than HPC such as resource sharing and load balancing. We will extend AMGE to implement system-wide policies that dynamically grow/shrink the GPU resources allocated to the different applications in the node.
Another type of platform that can benefit from our work is mobile devices, as they integrate CPU cores and accelerators. Thanks to HPE and AMGE, applications are able to run seamlessly on GPU systems with both integrated or separate memory subsystems. We will compare the execution efficiency that can be achieved on both systems.

Appendix A

Publications

A.1 Conference papers

- *"Automatic Parallelization of Kernels in Shared-Memory Multi-GPU Nodes"*. Javier Cabezas, Isaac Gelado, Lluís Vilanova, Thomas B. Jablin, Nacho Navarro, Wen-mei Hwu. In Proceedings of The 29th International Conference on Supercomputing (ICS 2015). Newport Beach, CA. June 2015.

- *"Automatic execution of single-GPU computations across multiple GPUs"*. Javier Cabezas, Isaac Gelado, Lluís Vilanova, Thomas B. Jablin, Nacho Navarro, Wen-mei Hwu. In Proceedings of The 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT 2014). Edmonton, Canada. August 2014. Short Paper.

A.2 Journal papers

- *"Runtime and Architecture Support for Efficient Data Exchange in Multi-Accelerator Applications"*. Javier Cabezas, Isaac Gelado, John Stone, Nacho Navarro, David Kirk, Wen-mei Hwu. In IEEE Transactions on Parallel and Distributed Computing. 2014.

- *Assessing Accelerator-based HPC Reverse Time Migration.* Mauricio Araya-Polo, Javier Cabezas, Mauricio Hanzich, Miquel Pericàs, Félix Rubio, Isaac Gelado, Muhammad Shafiq, Enric Morancho, Nacho Navarro, Eduard Ayguadé, Jose María Cela and Mateo Valero. In

IEEE Transactions on Parallel and Distributed Systems (TPDS). January 2011.

A.3 Workshops

- *"GPU-SM: Shared Memory Multi-GPU Programming"*. Javier Cabezas, Marc Jordà, Isaac Gelado, Nacho Navarro, Wen-mei Hwu. In Proceedings of the 8th Workshop on General Purpose Processor Using Graphics Processing Units. San Francisco, USA. February 2015.

- *"High Performance Reverse Time Migration on GPU"*. Javier Cabezas, Isaac Gelado, Enric Morancho, Nacho Navarro and José María Cela. In proceedings of the XIII Workshop en Sistemas Distribuidos y Paralelismo (Workshop on Distributed Systems and Paralelism). Santiago de Chile, Chile. November 2009.

A.4 Side publications

- *"Enabling Preemptive Multiprogramming on GPUs"*. Ivan Tanasic, Isaac Gelado, Javier Cabezas, Álex Ramírez, Nacho Navarro, Mateo Valero. In Proceedings of the The 41st International Symposium on Computer Architecture (ISCA 2014). Minnesota, USA. June 2014.

- *"Auto-tuning of data communication on Heterogeneous Systems"*. Marc Jordà, Ivan Tanasic, Javier Cabezas, Lluís Vilanova, Isaac Gelado, Nacho Navarro. In Proceedings of the IEEE Seventh International Symposium on Embedded Multicore/Many-core SoCs (MCSoC 2013). Tokyo, Japan. September 2013.

- *"Comparison based sorting for systems with multiple GPUs"*. Ivan Tanasic, Lluís Vilanova, Marc Jordà, Javier Cabezas, Isaac Gelado, Nacho Navarro, Wen-mei Hwu. In Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units. Houston, USA. March 2013.

- *"An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems"*. Isaac Gelado, Javier Cabezas, Nacho Navarro, John E. Stone, Sanjay Patel and Wen-mei W. Hwu. In Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV). Pittsburgh, USA. March 2010.

- *"Linux Kernel compaction through cold code swapping"*. Dominique Chanet, Javier Cabezas, Enric Morancho, Nacho Navarro and Koen De Bosschere. In Transactions on High-Performance Embedded Architectures and Compilers (HiPEAC). July 2007.

- *"The Cost of IPC: an Architectural Analysis"*. Isaac Gelado, Javier Cabezas, Lluís Vilanova and Nacho Navarro. In The Third Workshop on the Interaction between Operating Systems and Computer Architecture. San Diego (California), USA. June 2007.

- *"Building a Global System View for Optimization Purposes"*. Ramon Bertran, Marisa Gil, Javier Cabezas, Víctor Jiménez, Lluís Vilanova, Enric Morancho and Nacho Navarro. In The Second Workshop on the Interaction between Operating Systems and Computer Architecture. pp. 18-25. Boston (Massachusetts), USA. June 2006.

Bibliography

[1] AMD Graphics Core Next (GCN) Architecture. Technical report, AMD Corporation, 2012. www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.

[2] APU 101: All about AMD Fusion Accelerated Processing Units. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/apu101.pdf, 2012. AMD Corporation.

[3] C++ AMP: C++ Accelerated Massive Parallelism. https://msdn.microsoft.com/en-us/library/hh265137.aspx, 2012. Microsoft.

[4] NVIDIA GeForce GTX 680: The fastest, most efficient GPU ever built. Technical report, NVIDIA Corporation, 2012. http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf.

[5] NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect, 2012. NVIDIA Corporation.

[6] High Bandwidth Memory (HBM) DRAM. https://www.jedec.org/standards-documents/docs/jesd235, 2013. JEDEC.

[7] CUBLAS Library User Guide. http://docs.nvidia.com/cuda/cublas/index.html, 2014. NVIDIA Corporation.

[8] CUBLAS-XT. http://docs.nvidia.com/cuda/cublas/index.html, 2014. NVIDIA Corporation.

[9] CUSP: A Library for Sparse Linear Algebra. http://cusplibrary.github.io/, 2014. NVIDIA Research.

[10] Global Memory for ACcelerators. https://code.google.com/p/adsm/, 2014. Barcelona Supercomputing Center and University of Illinois at Urbana-Champaign.

[11] NVBLAS. http://docs.nvidia.com/cuda/nvblas, 2014. NVIDIA Corporation.

[12] NVIDIA GeForce GTX 980: Featuring Maxwell, The Most Advanced GPU Ever Made. Technical report, NVIDIA Corporation, 2014. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF.

[13] NVIDIA NVLINK high-speed interconnect. http://www.nvidia.com/object/nvlink.html, 2014. NVIDIA Corporation.

[14] Tegra K1 Next-Gen Mobile Processor. http://www.nvidia.com/object/tegra-k1-processor.html, 2014.

[15] The Matrix Market Top Ten. http://math.nist.gov/MatrixMarket/extreme.html, 2014. US National Institute of Standards and Technology.

[16] HSA Platform System Architecture Specification 1.0. http://www.hsafoundation.com/?ddownload=4944, 2015. HSA Foundation.

[17] AccelerEyes. *ArrayFire*, 2012. www.arrayfire.com/docs.

[18] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 2–13, New York, NY, USA, 1995. ACM.

[19] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: a processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture.*, pages 104–114, May 1990.

[20] Jennifer M. Anderson, Saman P. Amarasinghe, and Monica S. Lam. Data and computation transformations for multiprocessors. In *Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '95, pages 166–178, New York, NY, USA, 1995. ACM.

[21] Mauricio Araya-Polo, Javier Cabezas, Mauricio Hanzich, Miquel Pericàs, Fèlix Rubio, Isaac Gelado, Muhammad Shafiq, Enric Morancho,

Nacho Navarro, Eduard Ayguadé, José M. Cela, and Mateo Valero. Assessing Accelerator-Based HPC Reverse Time Migration. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):147–162, 2011.

[22] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.

[23] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An extension of the StarSs programming model for platforms with multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.

[24] R. Babich, M. A. Clark, B. Joó, G. Shi, R. C. Brower, and S. Gottlieb. Scaling Lattice QCD Beyond 100 GPUs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 70:1–70:11, New York, NY, USA, 2011. ACM.

[25] Amnon Barak and Amnon Shiloh. The MOSIX Virtual OpenCL (VCL) Cluster Platform. In *Proceedings of the Intel European Research and Innovation Conference*, Oct 2011.

[26] Richie D. Baud, Robert H. Peterson, G. Ed Richardson, Leanne S. French, Jim Regg, Tara Montgomery, T. Scott Williams, Carey Doyle, and Mike Dorner. Deepwater Gulf of Mexico 2006: America's Expanding Frontier. Technical Report MMS 2002-021, 2002.

[27] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[28] Laura Susan Blackford, J. Choi, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker. ScaLAPACK: a portable linear algebra library for distributed memory computers - design issues and performance. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing*, SC '96, Washington, DC, USA, 1996. IEEE Computer Society.

[29] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. Simple but Effective Techniques for NUMA Memory Management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 19–31. ACM, 1989.

[30] William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. NUMA policies and their relation to memory architecture. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS IV, pages 212–221, New York, NY, USA, 1991. ACM.

[31] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM.

[32] Bryan Catanzaro, Alexander Keller, and Michael Garland. A decomposition for in-place matrix transposition. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 193–206, New York, NY, USA, 2014. ACM.

[33] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[34] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an Object-oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[35] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC '09*, pages 44–54. IEEE Computer Society, 2009.

[36] Cray Inc. *Chapel Language Specification. Version 0.96*, 2014. `http://chapel.cray.com/spec/spec-0.96.pdf`.

[37] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.

[38] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 381–394. ACM, 2013.

[39] Jose Duato, Antonio José Peña, Federico Silla, Juan C. Fernández, Rafael Mayo, and Enrique S. Quintana-Ortí. Enabling cuda acceleration within virtual machines using rcuda. In *Proceedings of the 18th International Conference on High Performance Computing (HiPC)*, pages 1–10, Dec 2011.

[40] Jose Duato, Antonio José Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS)*, pages 224–231, June 2010.

[41] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. Compiling a High-level Language for GPUs: (via Language Support for Architectures and Compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 1–12, New York, NY, USA, 2012. ACM.

[42] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[43] Ondrej Fialka and Martin Cadik. FFT and Convolution Performance in Image Filtering on GPU. In *Tenth International Conference on Information Visualization, 2006.*, IV 2006, pages 609–614, July 2006.

[44] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily

Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, 28(4):13–27, July 2008.

[45] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proceedings of the fifteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XV, pages 347–358, New York, NY, USA, 2010. ACM.

[46] Peter N. Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture. Technical report, NVIDIA Corporation, 2009. http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_Nvidia's_Fermi-The_First_Complete_GPU_Architecture.pdf.

[47] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[48] IMPACT Group. Parboil benchmark suite. http://impact.crhc.illinois.edu/parboil.php, 2012.

[49] Intel Corporation. *Ivy Bridge Archictecture*, 2011.

[50] Intel Corporation. *Intel Xeon Processor E5-1600/2400/2600/4600 (E5-Product Family) Product Families. Datasheet- Volume 2*, 2012. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-1600-2600-vol-2-datasheet.pdf.

[51] Thomas B. Jablin, James A. Jablin, Prakash Prabhu, Feng Liu, and David I. August. Dynamically Managed Data for CPU-GPU Architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 165–174, New York, NY, USA, 2012. ACM.

[52] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic CPU-GPU communication management and optimization. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 142–151, New York, NY, USA, 2011. ACM.

[53] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, July 2005.

[54] Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *Proceedings of the 2005 workshop on Memory system performance*, pages 36–43. ACM, 2005.

[55] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class GPU Resource Management in the Operating System. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 37–37, Berkeley, CA, USA, 2012. USENIX Association.

[56] The Khronos Group Inc. *The OpenCL Specification*, 2013.

[57] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 277–288, New York, NY, USA, 2011. ACM.

[58] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Emmett Witchel, Amir Wated, and Mark Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 201–216, Berkeley, CA, USA, 2014. USENIX Association.

[59] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. *The high performance Fortran handbook.* MIT Press, Cambridge, MA, USA, 1994.

[60] Kiyoshi Kurihara, David Chaiken, and Anant Agarwal. Latency tolerance through multithreading in large-scale multiprocessors. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 91–101. IPS Press, 1991.

[61] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 245–256, Piscataway, NJ, USA, 2013. IEEE Press.

[62] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[63] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 101–110, New York, NY, USA, 2009. ACM.

[64] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25:63–79, 1992.

[65] Zoltan Majo and Thomas R. Gross. Matching memory access patterns and data placement for NUMA systems. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 230–241, New York, NY, USA, 2012. ACM.

[66] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. Feedback-directed page placement for ccNUMA via hardware-generated memory traces. *Journal of Parallel and Distributed Computing*, 70(12):1204 – 1219, 2010.

[67] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, pages 57–68, Aire-la-Ville, Switzerland, 2002. Eurographics Association.

[68] Paulius Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, New York, NY, USA, 2009. ACM.

[69] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques.*, pages 261–270, Sept 2009.

[70] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: A portable "shared-memory" programming model for distributed memory computers. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, SC '94, pages 340–349. IEEE Computer Society Press, 1994.

[71] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. Leveraging Transparent Data Distribution in OpenMP via User-Level Dynamic Page Migration. In *High Performance Computing, Third International Symposium, ISHPC 2000, Tokyo, Japan, October 16-18, 2000. Proceedings*, pages 415–427, 2000.

[72] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. User-level dynamic page migration for multiprogrammed shared-memory multiprocessors. In *Proceedings of 2000 International Conference on Parallel Processing*, ICPP 2000, pages 95–103, 2000.

[73] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-D FFT Library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 30:1–30:10, New York, NY, USA, 2009. ACM.

[74] NVIDIA Corporation. *CUDA C Programming Guide*, 2014. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[75] OpenACC-Standard.org. *The OpenACC Application Programming Interface Version 2.0*, 2013.

[76] PCI-SIG. *PCI Express Base 3.0 Specification*, 2010. `https://www.pcisig.com/specifications/pciexpress/base3/`.

[77] Gabriel Rivera and Chau-Wen Tseng. Tiling Optimizations for 3D Scientific Computations. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.

[78] Christopher J. Rossbach, Jon Currey, and Emmett Witchel. Operating systems must support GPU abstractions. In *HotOS'13*, pages 32–32. USENIX Association, 2011.

[79] Amit Sabne, Putt Sakdhnagool, and Rudolf Eigenmann. Scaling Large-data Computations on multi-GPU Accelerators. In *Proceedings of the*

27th International ACM Conference on International Conference on Supercomputing, ICS '13, pages 443–454, New York, NY, USA, 2013. ACM.

[80] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. vcuda: Gpu-accelerated high-performance computing in virtual machines. *IEEE Transactions on Computers*, 61(6):804–816, June 2012.

[81] Albert Sidelnik, Saeed Maleki, Bradford L. Chamberlain, Maria J. Garzarán, and David Padua. Performance Portability with the Chapel Language. *Parallel and Distributed Processing Symposium, International*, 0:582–594, 2012.

[82] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a File System with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 485–498, New York, NY, USA, 2013. ACM.

[83] John E. Stone, David J. Hardy, Ivan S. Ufimtsev, and Klaus Schulten. GPU-accelerated molecular modeling coming of age. *Journal of Molecular Graphics and Modelling*, 29(2):116 – 125, 2010.

[84] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, 2012.

[85] John A. Stratton, Christopher I. Rodrigues, I-Jui Sung, Li-Wen Chang, Nasser Anssari, Geng (Daniel) Liu, Wen mei W. Hwu, and Nady Obeid. Algorithm and data optimization techniques for scaling to massively threaded systems. *IEEE Computer*, 45(8):26–32, 2012.

[86] Yonghe Sun, Fuhao Qin, Steve Checkles, and Jacques P. Leveille. 3-D prestack Kirchhoff beam migration for depth imaging. *Geophysics*, 65(5):1592 – 1603, 2000.

[87] I-Jui Sung, Juan Gómez-Luna, José María González-Linares, Nicolás Guil, and Wen-Mei W. Hwu. In-place transposition of rectangular matrices on accelerators. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 207–218, New York, NY, USA, 2014. ACM.

[88] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling Preemptive Multiprogramming on GPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 193–204, Piscataway, NJ, USA, 2014. IEEE Press.

[89] Ivan Tanasic, Lluís Vilanova, Marc Jordà, Javier Cabezas, Isaac Gelado, Nacho Navarro, and Wen-mei W. Hwu. Comparison based sorting for systems with multiple GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 1–11, New York, NY, USA, 2013. ACM.

[90] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-purpose Uses. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 325–335, New York, NY, USA, 2006. ACM.

[91] Vinod Tipparaju and Jeffrey S. Vetter. GA-GPU: extending a library-based global address space programming model for scalable heterogeneous computing systems. In *Proceedings of the 9th conference on Computing Frontiers*, CF '12, pages 53–64, New York, NY, USA, 2012. ACM.

[92] Jonas Tölke. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by NVIDIA. *Computing and Visualization in Science*, 13(1):29–39, 2010.

[93] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing*, IPDPS 2010, pages 1 –8, April 2010.

[94] Sean Treichler, Michael Bauer, and Alex Aiken. Language support for dynamic, hierarchical data partitioning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 495–514, New York, NY, USA, 2013. ACM.

[95] Carlos Villavieja, Vasileios Karakostas, Lluís Vilanova, Yoav Etsion, Álex Ramirez, Avi Mendelson, Nacho Navarro, Adrián Cristal, and Osman S. Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *2011 International Conference*

on Parallel Architectures and Compilation Techniques (PACT), pages 340–349, Oct 2011.

[96] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. *ACM Sigplan Notices*, 26(6):30–44, 1991.

[97] Michael Wolfe. Implementing the PGI Accelerator Model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 43–50, New York, NY, USA, 2010. ACM.

[98] Shucai Xiao, Pavan Balaji, James Dinan, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. Transparent accelerator migration in a virtualized gpu environment. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 124–131, Washington, DC, USA, 2012. IEEE Computer Society.

[99] Shucai Xiao, Pavan Balaji, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu chun Feng. Vocl: An optimized environment for transparent virtualization of graphics processing units. In *In Proceedings of the 1st Innovative Parallel Computing (InPar)*, 2012.

[100] Yili Zheng, Costin Iancu, Paul Hargrove, Seung-Jai Min, and Katherine Yelick. Extending Unified Parallel C for GPU computing. SIAM Conference on Parallel Processing for Scientific Computing, 2010.