NTNU
Norwegian University of
Science and Technology

Modeling Communication on Multi-GPU Systems

Daniele Spampinato

Master of Science in Computer Science
Submission date: July 2009
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

This project evaluates how multiple Graphical Processing Units (GPUs) may best be utilized to offload computations in HPC environments. In particular, we look at models for heterogeneous systems and develop a PDE SOR solver that is used to analyze communication patterns when doing boarder exchanges on multiple GPUs. Our implementation take advantage of the CUDA environment. Testing will be done on the HPC-LAB's new NVIDIA Tesla S1070 and/or other appropriate systems.

Assignment given: 25. January 2009
Supervisor: Anne Cathrine Elster, IDI

NTNU
Norwegian University of
Science and Technology

Master Thesis

Modeling Communication on Multi-GPU Systems

Daniele Giuseppe Spampinato
daniele.spampinato@gmail.com

Department of Computer and Information Science
Norwegian University of Science and Technology, Trondheim
(Norway)

July 2009

Supervisor
Dr. Anne C. Elster

HPC Research Group

Preface

Many people, directly or indirectly, influenced the production of this master thesis, and I would like to extend to them my grateful appreciation for their help and support.

I would like to thank my supervisor, Dr. Anne C. Elster, for introducing me into the world of High Performance Computing, and for proposing, together with Thorvald Natvig, the target of this work. I also feel enormously thankful to her for all the great experiences that I had the opportunity to take part in, such as Supercomputing '08 (Austin, TX), IPDPS '09 (Rome), and the IBM EMEA BSRE '09 (Cracow). A special thanks to Jan C. Meyer for his precious suggestions and patience. Our endless, ontological discussions were for me source of great inspiration. And irreplaceable was the entire HPC-Lab at IDI: Rune Hovland, Rune E. Jensen, smund Herikstad, Daniel Haugen, Olav Fagerlund, Eirik Ola Aknes, Owe Johansen, Henrik Hesland, Safrudin Mahic, smund Eldhuset, and Robin Eidissen. I always found in them a valuable help and lots of good ideas. I wish to thank NVIDIA for providing several GPUs to Dr. Anne C. Elster and her HPC-lab through her membership in the NVIDIAs professor partnership program.

This master thesis signs for me the end of the T.I.M.E. project in Norway. It has been the longest and most exciting adventure of my entire life. Impossible to recall all the significant moments and make a complete list of acknowledgments within the space of one page.

My gratitude goes first and foremost to my parents, Armando Spampinato and Giuseppina Rota. They made all this things happen, always supporting and trusting me, no matter what. To them, who dedicate their life to their family, I would like to dedicate this work.

Even though far from home, I have never felt far from all my dears at home, starting from my sister Beatrice, and all my closest friends, that I cannot wait to hug again.

Living in Trondheim was an experience that I will never forget. The Moholt Family has a special place in my heart: Maru, Seu Lki Cecilia, Alejandra, Maria, Ole, Jon, Stefano, Seth, Eva & Luka, Daniele, Francesco "Friz", and Thomas. Together with them, there is a long list of people that I had the fortune to meet during these two amazing years, starting from the fantastic flatmates in MA 28-1: Emily, Kelsy, Gwyneth, Ravan, Elizaveta, Lars, Trine Lise, Helena, to all the other people I had a great time with: Heather, Matt, Anna, Andrea, Vegar, Fabian, Raphael, Sendy, Charlotte, Piotr, Thomas, p. Dominique, p. Etienne, p. Al, Huy, Maria, Karoline, Benedikt, Pia, Sara, Peter, Piotr, Matthias, Charlie, Sabine, Nikolaas, Francesco, Leiv, and many many others: Thanks!

In Italy we have a nice proverb that states:

"Chi trova un amico trova un tesoro."

One who finds a friend, finds a treasure. It must be true. Never felt richer so far.

Dealing with two different countries one must expect a "certain degree" of bureaucracy. I also wish to thank here the two Offices of International Relations for helping me in struggling with that, especially Francesca Fogal, that from Milan has been able to survive my countless requests.

Trondheim, July 16, 2009

Daniele Giuseppe Spampinato

Abstract

Coupling commodity CPUs and modern GPUs give you heterogeneous systems that are cheap, high-performance with incredible FLOPS counts. Recent evolution of GPGPU models and technologies make these systems even more appealing as compute devices for a range of HPC applications including image processing, seismic processing and other physical modeling, as well as linear programming applications. In fact, graphics vendor such as NVIDIA and AMD are now targeting HPC with some of their products. Due to the power and frequency walls, the trend is now to use multiple GPUs on a given system, much like you will find multiple cores on CPU-based systems. However, increasing the hierarchy of resource wides the spectrum of factors that may impact on the performance of the system.

The lack of good models for GPU-based, heterogeneous systems also makes it harder to understand which factors impact performance the most. The goal of this thesis is to analyze such factors by investigating and benchmarking NVIDIA's multi-GPU solution, their recent NVIDIA Tesla S1070 Computing System. This system combines four T10 GPUs making available up to 4 TFLOPS of computational power. Based on a comparative study of fundamental parallel computing models and on the specific heterogeneous features exposed by the system, we define a test space for performance analysis. As a case study, we develop a red-black, SOR PDE solver for Laplace equations with Dirichlet boundaries, well known for requiring constant communication in order to exchange neighboring data. To aid both design and analysis, we propose a model for multi-GPU systems targeting communication between the several GPUs.

The main variables exposed by the benchmark application are: domain size and shape, kind of data partitioning, number of GPUs, width of the borders to exchange, kernels to use, and kind of synchronization between the GPU contexts. Among other results, the framework is able to point out the most critical bounds of the S1070 system when dealing with applications like the one in our case study. We show that the multi-GPU system greatly benefits from using all its four GPUs on very large data volumes. Our results show the four GPUs almost four times faster than a single GPU, and twice as fast as two. Our analysis outcomes also allow us to refine our static communication model, enriching it with regression-based predictions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The strong interest of the scientific community in developing computational science applications that utilize graphics hardware, has favored a new trend in GPU architecture development. During the current decade, many of the most important manufacturers have thus introduced new product lines targeting scientific computation. In the course of few years, we reached the point where off-the-shelf workstations can combine multicore CPUs and graphics hardware providing performance results comparable to those of some supercomputers in the recent past.

It has been shown that the performance obtained by recent GPUs can get very close to the performance shown by the slowest Top-500 supercomputer only four years ago and the fastest one around 10 years ago. To be convinced, we may just take a look at the two graphs in Figure 1.1 and 1.2. The first one shows the per-

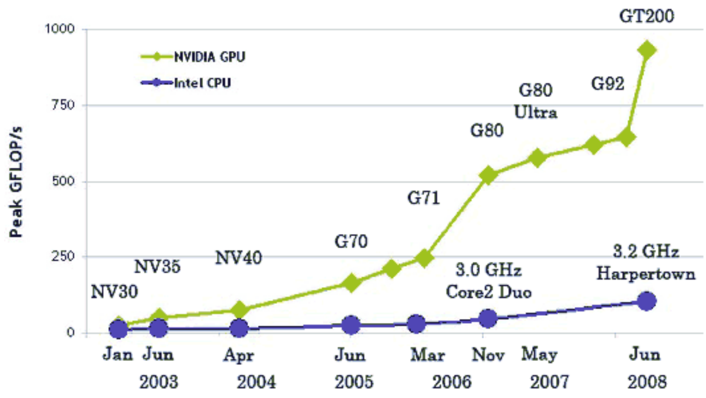


Figure 1.1 – GPU performance trend. (With permission from *NVIDIA*)

formance trend of GPUs during the present decade. The second one performance trends of the first and last supercomputer in the Top500 list within a larger period. We notice how the performance obtained by recent GPUs can get very close to the performance shown by the slowest supercomputer four years ago and the fastest one around 10 years ago.

Recent evolution of GPGPU models and technologies make these system even more appealing as compute devices for a range of HPC applications including image processing, seismic processing and other physical modeling as well as linear programming applications [27].

If providing a computing node with a GPU accelerator can improve performance, even better results should be achievable using more the one GPU per CPU. Multi-GPU systems increase resources and computing capabilities per node, offering the opportunity to leverage even more speed. An example of a recent multi-GPU system is the NVIDIA Tesla S1070 Computing System, which combine several GPUs to make available up to 4 TFLOPS.

However, increasing the hierarchy of resources issues new challenges to the developers, widening the spectrum of factors that may impact on the performance of a multi-GPU system.

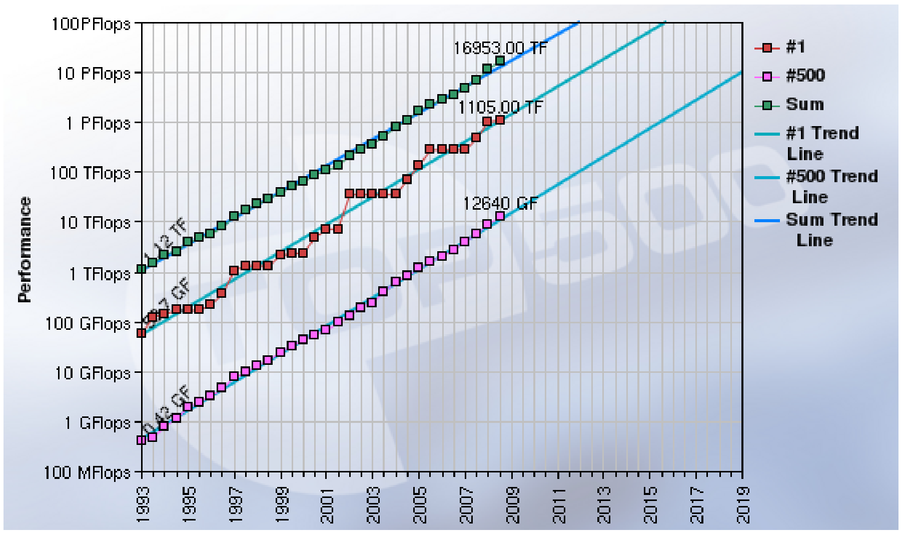


Figure 1.2 – Performance trends in the top500 list. (With permission from *top500.org*)

1.1 Thesis Goal

The absence of specific models for GPU-based, heterogeneous systems makes it harder to understand what factors are the most in influential in improving or worsening performance. The aim of this work is to investigate such factors in the specific context of the NVIDIA multi-GPU solution, the Tesla S1070 platform. Aside from the intrinsic issues caused by graphics technologies, we will also consider some important models of parallel systems in order to identify some common properties that can help us in our study. Communication is always a relevant aspect when dealing with distributed resources. By designing a benchmark framework around the SOR PDE solver, an applications that constantly requires inter-GPU communication, we are able to develop better a multi GPU model.

1.2 Outline

Chapter 2 presents the technological background, introducing the recent GPU technologies that will support our study. We will focus on NVIDIA's recent technologies, describing in particular the Tesla architecture processing model, the CUDA programming model, and multi-GPU programming.

Chapter 3 discusses parallel models. It analyzes the shared-memory and the message-passing models considering possible analogies with multi-GPU systems. These systems are then analyzed in order to point out the main challenges issued by the GPU programming model.

Chapter 4 contains the mathematical background necessary to understand the differential equation solver selected as the benchmark application.

Chapter 5 describes the benchmark test space, which dimensions are related to factors that may impact on performance. The chapter then presents the benchmark application. It highlights the most important decisions at design and implementation level.

Chapter 6 shows and discusses the results obtained running the benchmark. Results are collected using different test configurations to analyze how these different configurations impact the performance of the multi-GPU system used in our experiments.

Chapter 7 summarizes the project conclusions and suggests some future work directions.

Chapter 2

Multi-GPU Computing

Graphics hardware is now about forty years old. It was initially developed to support compute-intensive applications such as computer-aided design (CAD) and flight simulation [4]. The last generation of graphics processing units (GPUs) consists of highly parallel, multithreaded, manycore processors. Their large number of streaming processors are well suited for fine-grained, data-parallel workloads, consisting of thousands of independent threads operating concurrently.

Since late 70s, an always larger number of numerical applications are supported by graphics hardware, showing promising results in different scientific fields, such as linear algebra, data compression, database management, and financial services [4, 25, 6, 15]. Normally, when referring to such non-graphics employments of the GPU, it is typical to use the expression general purpose computation on GPUs, or shortly GPGPU[1]. However, the concept of general purpose programming applied to a GPU cannot be confused with the one associated to a normal CPU. Owens *et al.* [25] gives a concise description of the characteristics that an application must feature to successfully map onto a GPU. In particular, it has to be an application with large computational requirements and highly parallelizable, where the throughput is more important than latency.

In the following sections we will give an insight into the processing and programming models exposed by these recent technologies.

2.1 The NVIDIA Tesla Computing Architecture

NVIDIA Corporation[2] is currently one of the world's leading manufacturers of graphics technologies. NVIDIA releases GPUs targeting the most computation demanding fields, such as gaming, professional graphics processing, and high performance computing.

Since November 2006, NVIDIA's GPUs are based on the Tesla unified graphics and compute architecture, and since February 2007 they are provided with the

[1] http://www.gpgpu.org
[2] http://www.nvidia.com

CUDA programming environment to simplify many-core programming. In the official programming guide [23] the NVIDIA Tesla architecture is briefly but completely described in the following way: a set of SIMT multiprocessors with on-chip shared memory. To understand the Tesla's processing model, let us deepen the concepts contained into the previous definition.

2.1.1 Streaming Multiprocessors

The Tesla architecture is built around a scalable array of multithreaded streaming multiprocessors (SMs). A multiprocessor consists of eight scalar processors (SPs), two special function units for transcendentals, a multithreaded instruction unit (MIU), and on-chip memory. A SM creates, manages, and executes concurrent threads in hardware with zero scheduling overhead. This is an important factor to allow very fine-grained decomposition of problems by assigning, for instance, one thread to each data element.

2.1.2 GPU Memories

On a GPU we can localize two distinct kinds of memories: on-chip and device memory. With on-chip memory, we refer to:

- A set of 32-bit registers per SP;
- A parallel scratchpad memory, better known as shared memory, per SM. Access times to shared memory are comparable to a L1-cache on a traditional CPU;
- A read-only constant cache shared by all the SPs used to speed up reads from the constant area in device memory;
- A read-only texture cache shared by all the SPs used to speed up reads from the texture region in device memory.

Device memory is a high-speed DRAM memory with higher latency and larger dimension than on-chip memory (typically hundreds of times slower and million times larger). Device memory is subdivided in four regions:

- a read-write, non-cached global area;
- a read-write, non-cached local area;
- a read-only, cached constant area;
- a read-only, cached texture area.

A couple of comments concerning the memory terminology. Device and global memory are, at this point, clearly not synonyms. Global implies the access pattern to that specific area and it is part of the device memory of the GPU. Similarly, local and shared memory are not the same concept. Local memory is part of the device memory (slow) while shared memory is on-chip (fast). Local memory is used by the compiler to keep anything the developer considers local to a thread but, that for some reason, does not fit into the registers of the SP where the thread is executed (e.g. large structures that would consume too many registers).

2.1.3 The SIMT Paradigm

The single instruction, multiple threads (SIMT) paradigm is a new paradigm introduced to manage properly the big amount of threads executable on a Tesla-based GPU. A key difference between the SIMT and the SIMD paradigm is in that a SIMT architecture has no vector width, and a single instruction is issued for and executed by multiple processing elements, supporting full utilization of the cores every time. In contrast, SIMD architectures operate at a reduced capacity when the input size is smaller than the vector dimension.

Threads are grouped in warps and mapped to the SPs. One thread is mapped to one SP. Warps are created, managed, and scheduled by the MIU. The number of threads per warp is normally a multiple of the number of SPs. Threads that belong to the same warp start together at the same program address, maintaining a total freedom to branch and execute independently.

Every instruction issue time, the MIU selects a warp that is ready to execute and issues the next instruction to the active threads of the warp. A warp executes one common instruction at a time, so full efficiency is realized when all threads of a warp agree on their execution path. However, threads are free to execute differently, but when this happens, performance risks to be seriously injured. The reason is that disagreements on the control flow force the threads' scheduler to serialize their execution. Once taken all the independent paths, the threads converge back to the next common step in the execution path. Of course, since warps are executed independently, serialization is a problem that might occur just for threads of the same warp. Branch divergences are not the only reason that may lead to threads serialization. If an instruction executed by a warp, regardless of its atomicity, writes to the same location in shared memory for more than one thread in the warp, writes are also serialized. This side effect is called bank conflict. More details about it are given in Section 3.2.1.

2.2 The CUDA Programming Model

Recent programming models for GPUs are the last stage of the evolution of the graphics pipeline. GPGPU developers were used to adopt such a model when implementing their solutions. Basically, we can describe the graphics pipeline as a directed flow of data between its input and its output. The input is provided as a set of triangles which vertices are processed in the first stage of the pipeline, the vertex processor, applying transformations such as rotations and translations. Then, the raster converts the results from the vertex processing to a collection of pixel fragments by sampling the triangles over a specified grid. Such fragments are then computed by the fragment processor, which major task is to compute the color of the several fragments related to each pixel. Texturing, when required, is applied at this point. Finally the framebuffer determines the final color of each pixel in the final image, usually by keeping the closest fragment to the camera for each pixel location. At the beginning the pipeline was conceived as a rigid structure, where the different stages were implemented as fixed functions.

Very soon, GPU designers realized that allowing flexibility would open the possibility to develop more complex effects. In this scenario the pipeline assumed a new connotation, and since the present decade it allows programmability at different level of the processing flow. Older GPU programs were often called shaders, and they were written with shader languages, generally an extension of traditional programming languages in order to support vertices, fragments and interfaces to the pipeline stages (e.g. Cg [16]). Figure 2.1 depicts a typical graphics pipeline with shader programs for vertex and fragment processing. The next step towards a

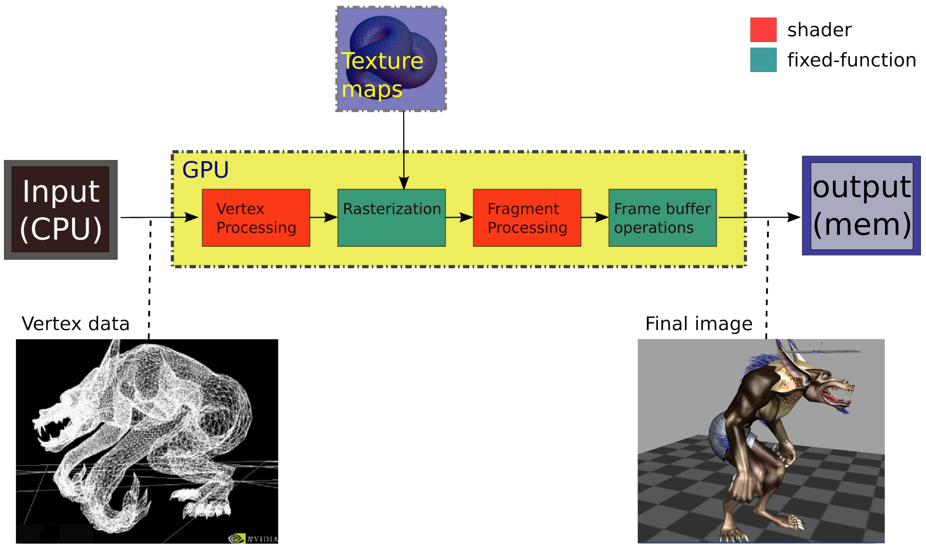


Figure 2.1 – The graphics pipeline.

modern GPGPU approach was the advent of the unified shader architecture. With the programmable pipeline described above, developers could take better advantage of the repartition of the GPU resources, but still with some unpleasant disadvantage, such as load unbalancing. In that context, the slowest stages burden the whole performance. In a unified shader architecture we find several shader cores able to operate at every level of the pipeline model. Each unified shader core can execute any type of shader and forward the result to another shader core (itself included), until the entire chain of shaders has been executed. The use of unified shader cores allows to allocate resources in a smarter way depending on the specific application, better managing load balancing.

The compute unified device architecture (CUDA) environment[3] presents a cutting-edge programming model well-suited for modern GPUs architectures [23]. NVIDIA developed this programming environment to fit to the processing model of the Tesla architecture, so to expose the parallel capabilities of GPUs to the developers.

[3]http://www.nvidia.com/object/cuda_home.html

2.2.1 A Heterogeneous Programming Model

CUDA maintains a separated view of the two main actors involved in the computation, namely the host and the device. The host is the one that executes the main program, while the device is the coprocessor. A typical scenario considers the CPU as the host and a the GPU as the coprocessor, but in general CUDA abstractions may be useful for programming other kinds of parallel systems [20].

Normally CUDA programs contain some pieces of code where intensive computation is required as shown in Figure 2.2. Such code is encapsulated in kernels and executed by the device. The same kernel is executed in parallel by several threads. The number of threads that execute a specific kernel is decided by the programmer and specified when invoking the kernel in the host program (See Section 2.2.3). From a memory point of view, CUDA assumes that host and device maintain their

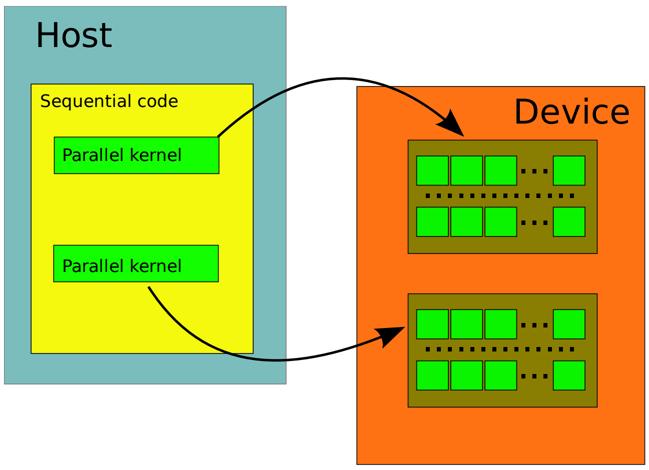


Figure 2.2 – Heterogeneous paradigm of execution.

own DRAM, respectively called host and device memory. The most basic example of CUDA processing flow is given in Figure 2.3, and it can be summarized in few elementary steps:

1. Allocate memory on the device;
2. Copy data from the host memory to the device memory;
3. Execute the kernel;
4. Copy data from the device memory back to the host memory;
5. Deallocate memory on the device.

2.2.2 The NVIDIA CUDA Software Stack

Figure 2.4 illustrates the NVIDIA CUDA software stack showing its main layers. A CUDA application can be built upon three main entities: the NVIDIA CUDA Driver API, the NVIDIA CUDA Runtime API, and the NVIDIA CUDA libraries.

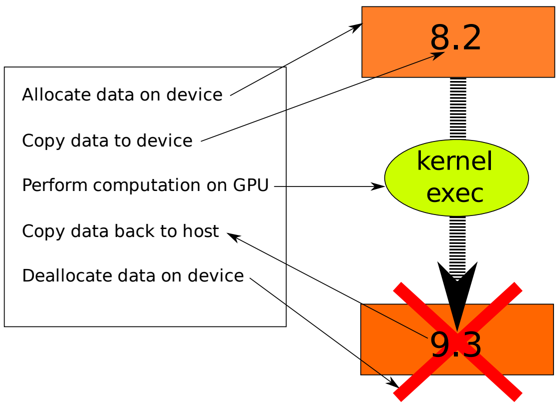


Figure 2.3 – Basic CUDA control flow.

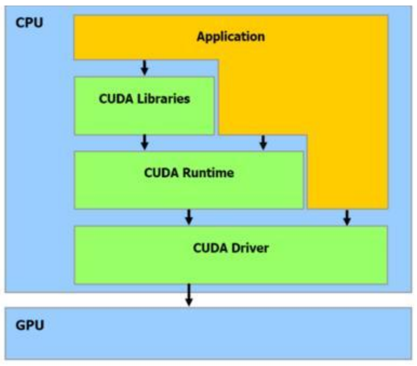


Figure 2.4 – The CUDA software stack. (With permission from *NVIDIA* [23])

The first two layers consist of a low-level Driver API and a higher level Runtime API implemented on top of the first one. Both the APIs are essential to proper manage threads, memory, and kernels invocation on the device. The Runtime API provides a more compact and intuitive interface to the device. On the other hand, even though the Driver API is harder to program, it offers deeper control and language-independence. Since these APIs do the same job at a different level, their use is mutually exclusive.

In the rest of the report every reference to kernel, thread, and memory management is intended through the Runtime API. The CUDA libraries contain mathematical routines of common usage written on top of the NVIDIA CUDA Runtime API, such as FFT and BLAS routines [21, 22].

2.2.3 Threads Organization

Scientific applications often have to cope with real problems, such as weather and climate forecasting, galaxies evolution, fluid simulation, protein folding, and so on. Applications such as those just mentioned are typically based on numerical methods that require to sample their problem domains in order to have a number of discrete data to elaborate.

To better match such mathematical models, the NVIDIA CUDA programming model expresses task and data parallelism through the threads hierarchy. A kernel is a portion of code executed in parallel by different threads. Threads are organized in one-, two-, or three-dimensional, equally-shaped threadblocks. This organization provides a natural way to work with multidimensional structures. Threads within the same threadblock can share data and synchronize themselves through intrinsic synchronization functions.

Multiple threadblocks are organized in one- or two-dimensional grids. Since threadblocks can be executed in any order, in parallel or not, they are required to execute independently. This independence requirement is the key to write scalable code, as it allows thread blocks to be scheduled in any order across any number of cores. While the number of threads per threadblock is limited by physical resources (Section 2.2.5), the number of threadblocks per grid is normally related to the size of the data to be processed.

The programmer has to know a priori the number of threads he wants to dedicate to a specific kernel, and he has to declare it in terms of grid and threadblock dimensions when calling the kernel:

```
kernel <<< dimGrid, dimBlock >>> (...list of parameters...)
```

The expression marked by the two triples of less-than and greater-than signs is called execution environment. Every kernel call is asynchronous, that means that the control returns to the CPU immediately, and it is normally executed by the GPU once all previous CUDA calls have completed. It is also possible to organize kernel executions on different streams, so making kernel calls independent from each other (see Section 3.2.1).

NVIDIA CUDA provides built-in variables that help identifying a number of useful information, like the index of a thread (threadblock) in a threadblock (grid) and the dimension of threadblocks and grids. In this way for instance, a single thread can compute the value of a specific element within a 3D matrix. The thread hierarchy together with such a fine control over the threads allows to define different levels of parallelism [20]. Independent threadblocks of a grid express coarse-grained data parallelism, while grids express coarse-grained task parallelism. Figure 2.5 enriches the execution model of a CUDA program with the threads organization just described.

2.2.4 Memory Organization

Every thread has its own local memory. Aside that, threads can use also shared, global, texture and constant memory. Data allocated in shared memory is visible to and accessible by all the threads within the same threadblock. Such data has the same lifetime as the threadblock itself. The global, texture, and constant memory are persistent across kernel invocations by the same application. Figure 2.6 exemplifies the concepts.

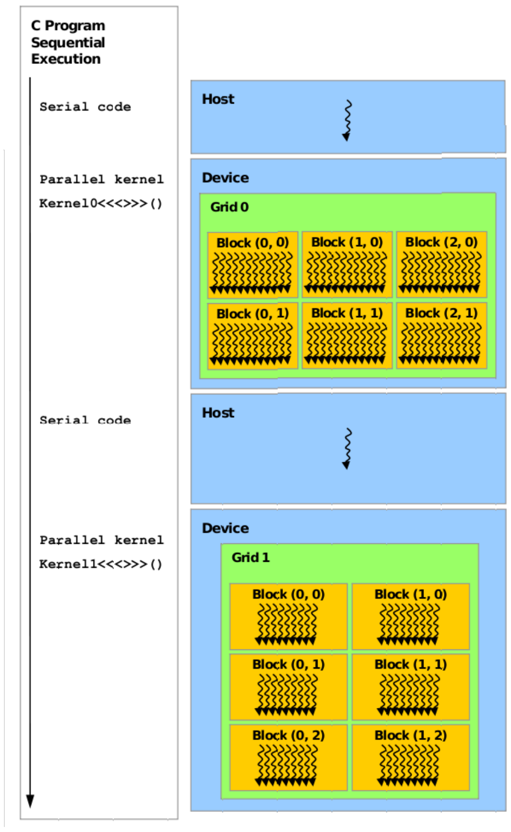


Figure 2.5 – Heterogeneous programming with CUDA. (With permission from *NVIDIA* [23])

2.2.5 Mapping to the Tesla Architecture

When a kernel is invoked, the several threadblocks that compose the grid are enumerated and distributed to SMs with available execution capacity. Every SM can execute several threadblocks. Threads are grouped in warps to be executed. The way a threadblock is split into warps is predefined and always the same: each warp contains threads of consecutive IDs with the first warp containing thread 0. If all the threads of a warp agree on the same execution path than we can estimate the number of clock cycles the GPU needs to execute the whole warp as S_w/N_{SP}, where S_w is the warp size and N_{SP} is the number of SP per SM.

The number of threadblocks a SM can process at once is related to the configuration of the specific launched kernel, and in particular to the amount of registers per thread and shared memory per threadblock requested. This because registers and shared memory are split among the threadblocks associated to the SM. A critical point is that to execute a kernel, a GPU requires enough registers and shared memory per SM to process at least one threadblock, otherwise the kernel invocation would fail.

From a memory point of view, CUDA shared, global, texture, and constant memory naturally fit to the respective Tesla memory areas. From the host, through

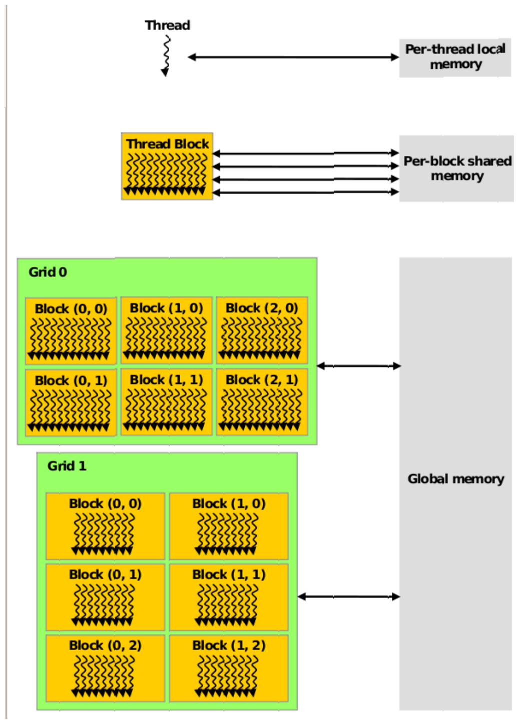


Figure 2.6 – The CUDA memory hierarchy. (With permission from *NVIDIA* [23])

the runtime library, it is possible to allocate memory on the device as linear memory or CUDA arrays. The first allocation method allows a pointer-based management of the memory. CUDA arrays are opaque objects designed and optimized for texture fetching.

2.2.6 Compute Capability

Every NVIDIA GPU that can be used in parallel compute mode, is characterized by a compute capability number. Compute capability numbers are defined by a major revision number and a minor revision number. The major revision number indicates the core architecture, while the minor number corresponds to incremental improvements of the core architecture with new features.

In Appendix A, the most relevant features of devices of compute capability 1.3 are summarized. The SP's clock frequency and the total amount of device memory can vary depending on the specific device and can be queried at runtime. The complete specification for the all set of available compute capabilities can be found in [23].

2.2.7 Floating-Point Support

All the GPUs of compute capability 1.3 can operate with both single- and double-precision floating-point values. However, GPUs belonging to the GeForce GTX 200 family are designed for gaming, where the speed is often more valuable than precision. Those GPUs present some deviations from the IEEE 754 standard for single-precision floating-point values, such as absence of denormalized numbers and smaller set of rounding modes. On the other hand, GPUs designed for HPC completely adhere to the IEEE standard, trying to fulfill the precision requirements typical of scientific applications.

2.3 The NVIDIA Tesla S1070 Computing System

The NVIDIA Tesla S1070 Computing System is a 1U rack-mount system equipped with four Tesla T10 GPUs. Figure 2.7 shows a schematic representation of the system. The Tesla T10 has 240 processing cores working either at 1.296 GHz (-400 configuration) or at 1.44 GHz (-500 configuration). The cores are grouped in 30 SMs. Each core is able to issue up to 3 FLOP per cycle in single precision, i.e. a *multiply* concurrently to a *multiply-add*. A theoretical estimation of the processing

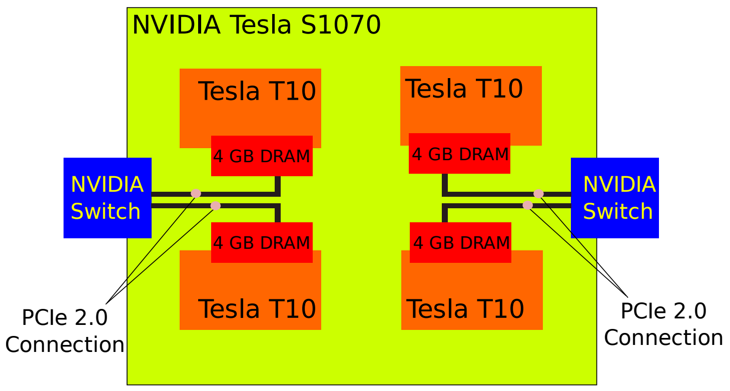


Figure 2.7 – NVIDIA Tesla S1070 Computing System Architecture.

power of the system is

$$\begin{aligned}
&4\,\text{GPUs} * (\text{FLOP/cycle} * \text{Frequency} * \#\text{Cores})_{\text{-400 conf.}} = \\
&\quad 4 * (3 * 1.296 * 240)\ \text{GFLOPS} = 3.732\ \text{TFLOPS in single precision.} \\
&4\,\text{GPUs} * (\text{FLOP/cycle} * \text{Frequency} * \#\text{Cores})_{\text{-500 conf.}} = \\
&\quad 4 * (3 * 1.44 * 240)\ \text{GFLOPS} = 4.147\ \text{TFLOPS in single precision.}
\end{aligned}$$

In double precision, only one core per SM can issue two concurrent operations per cycle, that leads to a theoretical peak of

$$\begin{aligned}
&4\,\text{GPUs} * (\text{FLOP/cycle} * \text{Frequency} * \#\text{SMs})_{\text{-400 conf.}} = \\
&\quad 4 * (2 * 1.296 * 30)\ \text{GFLOPS} \approx 311\ \text{GFLOPS in double precision.} \\
&4\,\text{GPUs} * (\text{FLOP/cycle} * \text{Frequency} * \#\text{SMs})_{\text{-500 conf.}} = \\
&\quad 4 * (2 * 1.44 * 30)\ \text{GFLOPS} \approx 345\ \text{GFLOPS in double precision.}
\end{aligned}$$

From a memory point of view, every GPU is connected to 4 GB high speed DRAM, with a bandwidth of 102 GB/s. This gives to the system a capacity of 16 GB. The connection to the host passes through NVIDIA Switches and PCIe Host Interconnection Cards (HIC). A single PCIe 2.0 16x (or 8x) slot on the host is connected to two GPUs using an NVIDIA Switch and a PCIe HIC. Such a connection is able to provide a transfer rate up to 12.8 GB/s between the host node and the computing system. A configuration that fully exploits the computing system is given in Figure 2.8. Since power consumption is taking more and more attention today, we can

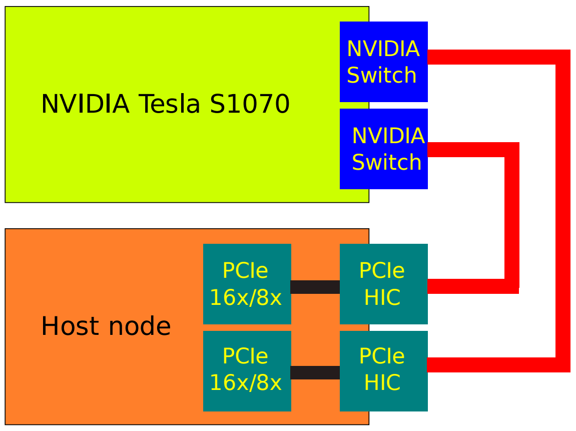


Figure 2.8 – S1070 full computing configuration.

also mention that the system consumes at most 800 W.

2.4 Programming Multiple GPUs

In a multi-GPU context, devices are enumerated progressively. The CUDA Runtime API gives the programmer the possibility to select the device where to execute the kernels. By default device 0 is used. The official CUDA guide [23] reports that a multi-GPU system is guaranteed to work only if the system is composed by the same type of GPUs, and if the Scalable Link Interface (SLI) mode is turned off. SLI is an NVIDIA solution for computer graphics. Basically, it allows two or more GPUs to work together to produce a single graphical output from different input images processed in parallel.

We could take into account at least two main ways to manage the different control flows associated to the GPUs, as depicted in Figure 2.9 for a two-GPUs system. A first way could be to send kernels in a sequential fashion. If the amount of work required to the first GPU is substantial it can be considered a valid solution. Otherwise, a second possibility is to exploit the parallelism offered by modern CPUs.

To use multiple CUDA context we can associate them to different threads, one for each GPU. For optimal performance, the number of CPU cores should not be less than then the number of GPUs in the system. Managing the threads could be done using different approaches. The lowest level one could be to implement an ad-hoc communication layer through system libraries, such as NPTL. Otherwise, existing libraries could be used, such as message-passing libraries adapted to perform shared memory communication [17].

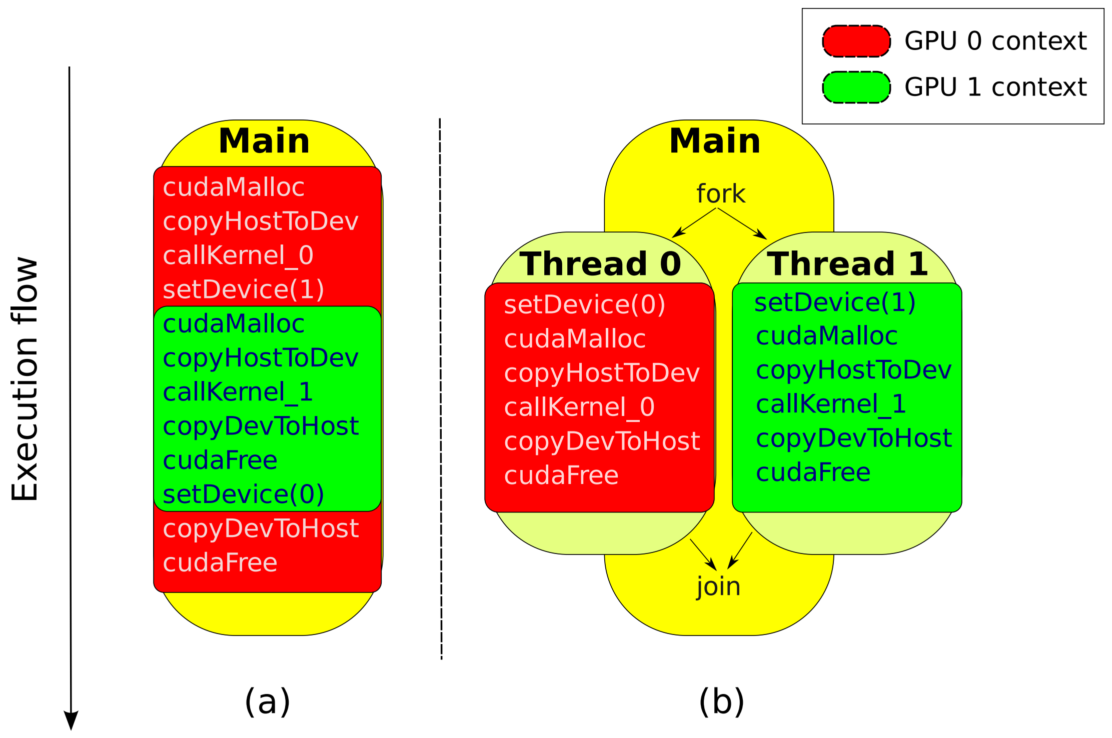


Figure 2.9 – Two possible approaches to program multiple GPUs: (a) sequential and (b) multithreaded.

Chapter 3

Parallel Computational Models

In this chapter, we analyze some important parallel computing systems looking for analogies with multi-GPU systems. The aim is to identify the factors that most impact on the performance of a multi-GPU system.

As argued in [19], parallel models often lack of connection to the real world, becoming powerless tools in terms of prediction capabilities. Nontheless, they do not loose their relevance when analyzing new architectures, because they help in focusing on the main characteristics exposed by the systems at issue.

In Section 3.1 we describe two fundamental types of parallel computers, i.e. the shared memory multiprocessor and the message-passing multicomputer. Following, we mention symmetrical multiprocessor (SMP) clusters, which are cost-effective computing platforms that mix shared memory and message-passing features. In Section 3.2, we analyze multi-GPU systems, highlighting their performance factors. Eventual similarities with more classical parallel models can be used to apply common solution patterns to the new architecture.

3.1 Parallel Models

There are multiple ways to build a parallel computing system. The two most popular models of parallel computation are shared memory multiprocessors and message-passing multicomputers. Based on these two basic models it is possible to build different kinds of computing systems, combining in different ways their architectural properties. SMP clusters are one such example where the computing platform combines elements from both the computing models.

3.1.1 Shared Memory Multiprocessor

A shared memory multiprocessor is composed by a set of processing units and a set of memory modules as represented in Figure 3.1. Processing units can be organized at different levels, such as chip-level (a.k.a. multicore CPU) or package-level. In a shared memory system, every single memory location is accessible by any of the processors. This is done implementing a single address space, which assigns to every memory location a unique address. In other words, the set of memory modules are

considered as a unique memory volume by the processors. Processors and memory are connected together through some form of interconnection network.
Which kind of interconnection is more opportune for a specific architecture, always depends on its features, such as dimension and cost. Typical interconnection networks are single buses shared by both processors and memory (for small systems), crossbar switches, or any other combination of the two. The kind of interconnection classifies the system depending on the memory access time. When each memory location is accessible to every processor with the same access time, the system has uniform memory access (UMA). On the other hand, when some of the processors require less time to access some memory location, the system has non-uniform access time (NUMA). To reduce communication in the network, it is possible to augment

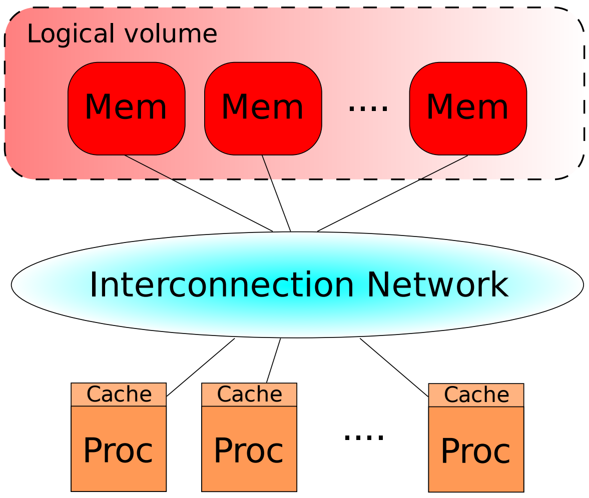


Figure 3.1 – A shared memory multiprocessor model.

the system with different level of caches to hold the contents of recently referenced memory. There are several ways to program a shared memory system. For the purpose of our work, we just focus on multithreaded programming, where every processor can execute one or more threads. For a broader treatment of shared memory programming methods one could refer to Wilkinson and Allen [29].

When programming a shared memory system, there a number of performance issues to take into consideration. We will now describe three important ones: threads synchronization, caching, and consistency.

Synchronization

Shared memory systems give the possibility to organize communication among the processors through memory. Since each memory location is accessible by all the processors, it is possible to use the memory to exchange data among them. If different threads are executed on physically different cores, real parallelism is achieved. Indeterministic scheduling of the threads requires the adoption of proper programming measures.

When accessing concurrently the same memory area without an accurate logic, it is possible to produce unexpected results. To prevent such problems threads must be coordinated through synchronization. Most relevant synchronization techniques include thread joins, locks, condition variables, semaphores, and monitors [28]. Consider a group of threads running in parallel some code (see Figure 3.2). If they access different memory locations, then they can be executed concurrently without risks. When some of them reference to the same memory space instead, a way to preserve data coherence is to introduce mutual exclusion on the critical section of the code. This can be done through one of the synchronization techniques mentioned above. However, the fact the just one thread can access the data at a time, requires the execution to be serialized, thus introducing latency in the computation.

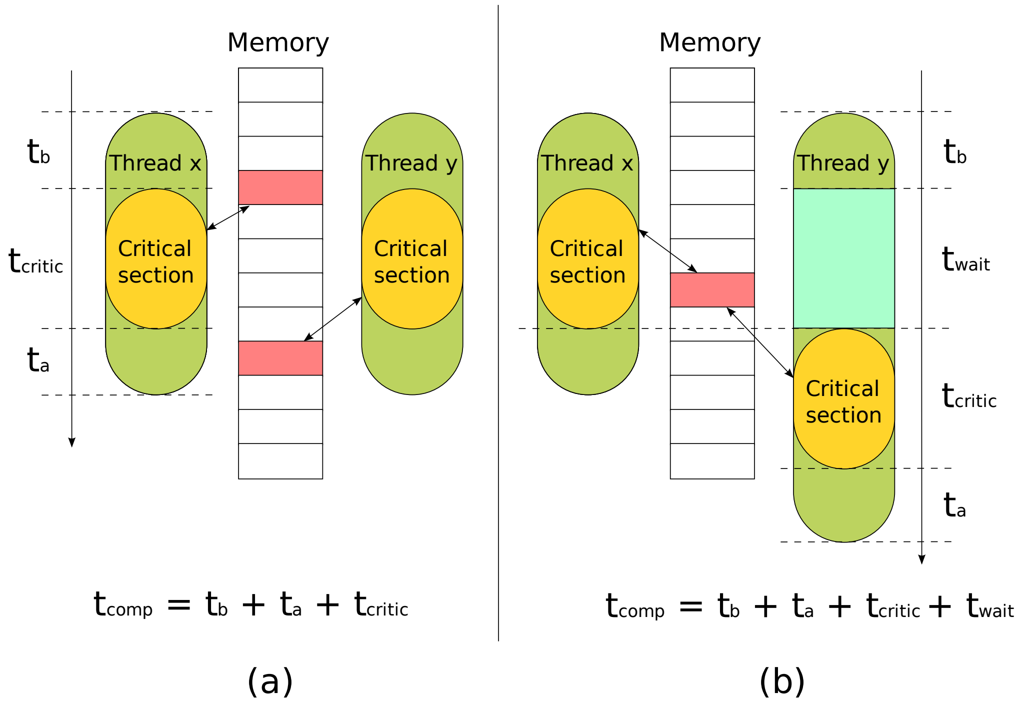


Figure 3.2 – Parallel execution: (a) critical sections accessing different memory locations, (b) critical sections working on the same data set.

Caching

Caches are intermediate memory storages used to deliver higher performance when accessing data or instructions. When accessing a variable or executing an instruction, instead of moving the single requested element, the memory subsystem transfers a block of data that contains it. This block is stored inside the cache, and if the next time the CPU issues that same element or one in the nearby, it can be served very quickly, obtaining the element from the cache.

Different levels of caches can be placed in between the main memory and the processing unit. This levels are characterized essentially by their position and dimension. Normally, on-chip storages have capacity in the order of kilobytes (L1 cache) and megabytes (L2/L3 cache). Other levels can be localized off-chip with a larger capacity but also a higher latency.

Caches are designed to be invisible to the developer, in the sense that one is not asked to consider them while developing his software. For this reason, they are not programmable elements. However, a cache-aware approach is very important in HPC, since a right use of the cache is able to provide order of magnitude differences in performance [12]. CPU vendors typically release information about the most important features (e.g. cache size, line size, number of ways, etc..) of the caches installed on their processors. Based on these specifications, developers can tune their memory access patterns speeding up their applications. Several methods exists that help optimizing the cache usage [8]. In a parallel environment, there are several issues that could significantly impact on performance.

A first expedient requires to take advantage of temporal and space locality. This can be done grouping instructions that are using same elements, and grouping instructions that access elements that are likely to be stored in a same memory block. When grouping instructions then, it is important to consider the access pattern involved. Often, the logical displacement of objects at the programming level does not reflect the physical displacement at a lower level in memory. For instance, this happens with arrays. Even though accessing a neighbor of an element in a matrix only requires to increment or decrement one the indexes, physically, the neighbor may be found dozens of locations away from the starting element.

When using caches in a shared memory system, another important objective is to reduce false sharing. The false sharing issue is related to the distributed nature of the system. Caching data in a parallel systems, introduces coherency requirements among the nodes' caches. Writing to memory locations stored in the cache of one of the nodes, causes all the blocks in the caches of other nodes to be invalidated or updated if they contain the same values.

Consistency

A distributed systems may be adherent to different models called consistency models, that express some properties of parallel executions' final results. One such models is called sequential consistency [14]. Sequential consistency is a consistency model often implemented by shared memory systems. According to the model, when running multiple concurrent programs in their respective orders, the final results has to be the same independently of the instructions interleaving. Some systems, to improve performance, slightly modify the concept of sequential. They allow some degree of out-of-order execution within a program, but always maintaining the sequential consistency of the final result.

3.1.2 Message-Passing Multicomputers

When the number of processing units in a parallel computing system increases dramatically, designing a proper shared memory architecture can be rather difficult for several reasons, such as processors-memory connection and cache/memory coherency. Alternatively, a parallel computing model could be composed by several computing elements where each of them is connected to its own memory module. Then, always through some kind of interconnection network, all the processor-memory pairs are connected to each other. Such computing architectures are called message-passing multiprocessors or multicomputers, since very often the processor-memory pairs are autonomous computing nodes.

A model of multicomputer is given in Figure 3.3. This kind of architecture

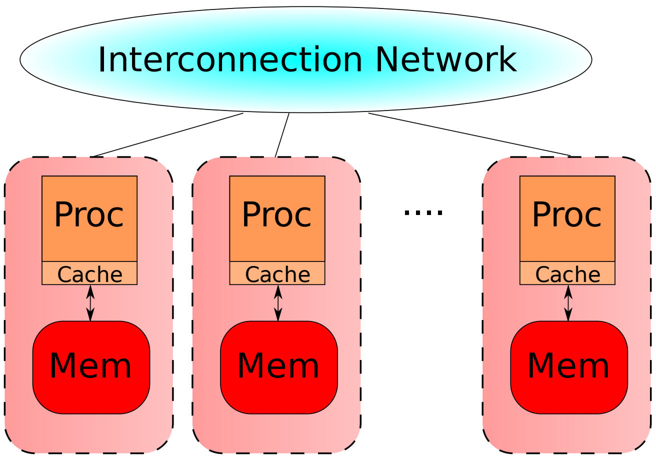


Figure 3.3 – A multicomputer model.

allows higher scalability and ease of construction, but changes completely the communication pattern. Instead of sharing data through a common memory volume, in a multicomputer context data must be transfered from a node to another, thus introducing new performance issues. A simple model for a parallel program is estimated in [29] considering its computation and communication time, which in general depend on the size of the problem and the number of processors:

$$T_{par} = T_{comp}(n, p) + T_{comm}(n, p) \; . \tag{3.1}$$

On a shared memory multiprocessor, the communication process is mainly composed by the synchronization overhead. In a message-passing multicomputer instead, we should model the cost of interprocess communication required to produce the final result. What the term t_{comm} contains depends on the specific communication flow required by the application. If we consider the whole communication time as composed by several communication flows among the processors related to different parts of the program, i.e.

$$T_{comm}(n, p) = \sum_i T^i_{comm}(n, p) \; , \tag{3.2}$$

we may describe the time required by the i^{th} communication flow as

$$T^i_{comm}(n, p) = k^i(n, p)(t_{startup} + w^i(n, p)t_{word}) \; . \tag{3.3}$$

The time $t_{startup}$ is the message latency, t_{word} is the time required to transfer one data word, and $w^i(n, p)$ is the number of data words to be transfered, which in general could depend on the problem and the system size. $k^i(n, p)$ is a general factor to represent a certain number of communication iterations required by the communication flow.

In a multicomputer, a very important objective is to improve performance trying to contain the communication complexity. Depending on the problem, different approaches must be developed to control the communication. However, there is a general principle which is worth remarking. Data transfer are always orders of magnitude greater than elementary operations, such as mathematical ones. In order to not waste time just waiting for data transfers to complete, many communication routines have their nonblocking counterparts. In this way, it is possible to do useful work, while at the same time, the message is moving towards its target (See Figure 3.4). This technique is known as latency hiding.

As for shared memory programming, also message-passing programming can be

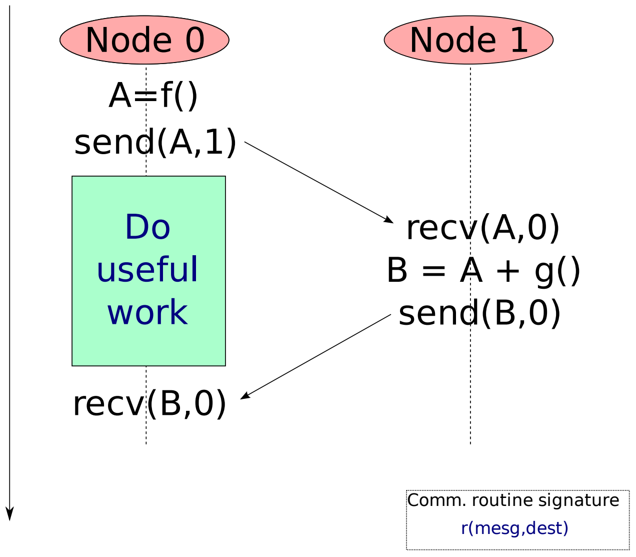


Figure 3.4 – Latency hiding.

done in different ways [29]. One common form, is to use a normal high-level programming language, such as C, together with message-passing libraries. The Message-Passing Interface (MPI) standard [1, 26] is very likely the most adopted standard interface for message-passing programming.

3.1.3 SMP Cluster Computing

A cluster of computers is a cost-effective example of multicomputer system. It represents a satisfying trade off between a single commodity computer and an expensive supercomputer in terms of costs and performance. Clusters themselves represent a large family of systems classified by different purposes and implementations.

One of them, called symmetrical multiprocessor (SMP) cluster, is especially interesting for our analysis, insofar as it has a computing model comparable, in our opinion, to the one of a multi-GPU system. A symmetrical multiprocessor is a shared memory multiprocessor with a numerical symmetry between the number of processors and the number of memory modules. When a number of SMPs are linked together, it gives birth to a new architecture called SMP cluster and depicted in Figure 3.5. For the sake of our discussion, The concept of symmetry is not a relevant feature. What is interesting to us, is the programming model that can be adopted on such a architecture. The outer set of SMP nodes could be programmed through message-passing, while multithreading can be used to manage the computation on every shared memory SMP.

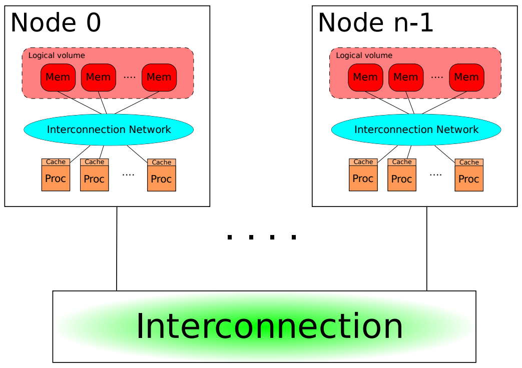


Figure 3.5 – An SMP cluster model.

3.2 Similarities between SMP Clusters and multi-GPU Systems

A multi-GPU computing architecture, as described in Chapter 2, has several characteristics that make it similar to the computing model of an SMP cluster. A first analogy comes from the system implementation. As shown in Figure 3.6, every GPU, which in turn is a highly multithreaded system, is linked to every other through the host system. This kind of interconnection, requires communication to setup cooperation in solving a common problem. In th previous chapter, we remarked how GPU systems are today able to compute with theoretical performance in the order of TFLOPS, transferring data between host and device with a rate in the order of GB/s. If we compare that with performance and transfer rate of typical cluster node, we observe that the theoretical instruction densities differ by a constant factor:

$$\frac{Per_{GPU}}{BW_{multi-GPU}} = \frac{TFLOPS}{GB/s} \approx k \cdot \frac{Per_{CPU}}{BW_{cluster}} = k \cdot \frac{GFLOPS}{MB/s} \,. \qquad (3.4)$$

In this section, we show how some performance issues analyzed in the previous section, can still hold for a multi-GPU system. Of course, because of the difference between a usual processor and a graphics processor, some of the concepts need a proper contextualization, posing sometimes new problematics to surmount.

3.2.1 Single GPUs as Shared Memory Nodes

A single NVIDIA Tesla computing architecture, contains hundreds of computing cores, organized in several SMs, connected to different levels of memory. The structure behind processors and memory, involves some new issues that must be carefully

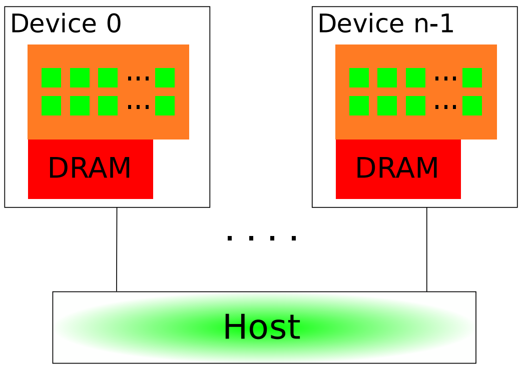


Figure 3.6 – Multi-GPU system as a network of GPUs.

considered. In Section 2.2.1, we introduced a basic CUDA control flow that represents a common host programming pattern. The next step is to understand how kernels can be organized, defining a basic programming pattern also for them. Figure 3.7 shows such a pattern. Every thread loads its own data in shared memory, so to have faster access to them. After that, it performs the required calculation on the data set and writes everything back to global memory. As communication through shared memory might be required by the threads of a same threadblock, synchronization is used to keep results coherent. At the level of a single GPU, an

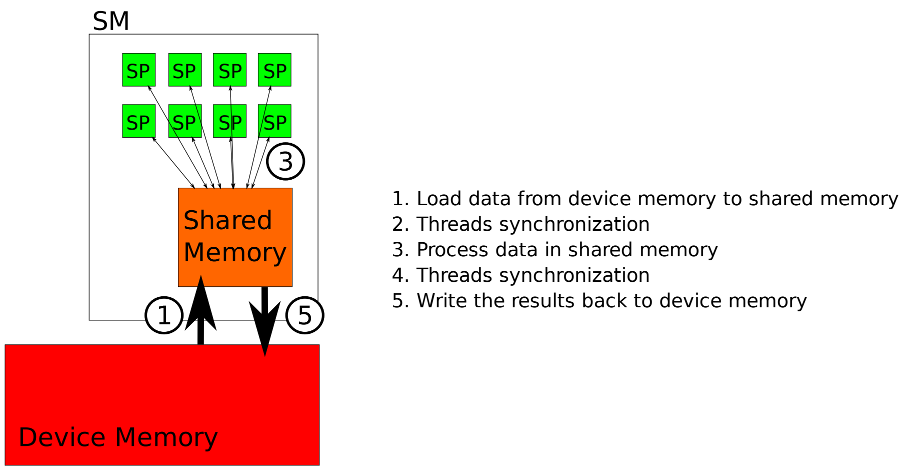

SM
SP
SP
SP
SP
SP
SP
SP
SP
3
Shared Memory
1
5
Device Memory
1. Load data from device memory to shared memory
2. Threads synchronization
3. Process data in shared memory
4. Threads synchronization
5. Write the results back to device memory

Figure 3.7 – Kernel programming pattern.

important step for increasing performance is to tune the algorithm in a way that it exposes as much data parallelism as possible. After that, the computing resources have to be allocated, mapping them to the problem using a proper configuration of the execution environment. Memory usage must be optimized at every level of the hierarchy, so to maximize the bandwidth. This is a very crucial step that requires to:

- Minimize the host-device communication. We will focus on this issue in Section 3.2.2 as part of the GPU-GPU communication;
- Use the device memory exploiting the different memory areas' capabilities;

- Use the right access pattern for each type of memory.

Finally, instructions must also be optimized, selecting the fastest ones. We hereby set about discussing these issues in more detail.

Execution Configuration

A good execution configuration should ensure an efficient mapping between the thread hierarchy and the parallelism exposed by the problem to solve. We already introduced execution configuration in Section 2.2.3. Here, we intend to discuss how grid structuring can influence performance. To reduce scheduling overhead, resources are split and allocated to each thread. Since resources are in a finite number, it is responsibility of the developer to choose wise dimensions for both grids and threadblocks, so to allocate resources minimizing wastes. If, for instance, at a certain point there are less active threadblocks than available SMs, the computing power of those SMs is wasted. In the same way if there are threads per threadblock that are not multiple of the warp size, a number of SPs would turn out unused. On the other hand, also oversize threadblocks are dangerous. If the number of registers required by a threadblock cannot fit to those available in a SM, the kernel invocation would not succeed.

A typical parameter that can lead to optimize the usage of the GPU is the so called multiprocessor occupancy. It is defined as the ratio of the number of active warps per multiprocessor to the maximum number of active warps,

$$\mathrm{SM}_{occ} = \frac{\mathrm{W}_{active}}{\mathrm{DEF}(\mathrm{W}_{max\_active})} \ . \tag{3.5}$$

The maximum number of active threads per SM, like many other parameters that depends on the specific hardware device, is normally specified by the compute capability of the GPU. Example of such parameters are given in Appendix A for devices of compute capability 1.3. In our notation, we identify the value of a parameter related to a specific compute capability number with the call to DEF($param$).

We can formalize such a ratio starting from those variables which definition is normally up to the developer: the number of threads per threadblock T_B, the number of registers per thread R_T and the requested amount of shared memory per threadblock Sh_{B_req}. A first set of parameters we can calculate are the number of warps, registers, and shared memory per threadblock. Warps and shared memory per threadblock are directly calculated as

$$W_B = \left\lceil \frac{T_B}{\mathrm{DEF}(T_W)} \right\rceil \ , \quad Sh_B = \lceil Sh_{B\_req}, \mathrm{DEF}(Sh_{alloc\_unit}) \rceil \ . \tag{3.6}$$

We indicate with $\lceil n, s \rceil$ the number n rounded up to a multiple of s. When $s = 1$ we simply use $\lceil n \rceil$. Analogous notation is used for the floor function. Values indicated with a notation similar to X_{alloc_unit} indicate the allocation unit sizes for those resources they refer to. So in our case, the number of warps per threadblock W_B is the calculated as the ratio of the number of threads per threadblock to the number of threads per warp allowed by the specific GPU. The number of registers per threadblock can be obtained as

$$R'_B = \frac{Warp}{Block} \cdot \frac{Thread}{Warp} \cdot \frac{Reg}{Thread} = \lceil W_B, \mathrm{DEF}(W_{alloc\_unit}) \rceil \cdot 32 \cdot R_T \, . \tag{3.7}$$

The value R'_B has to be rounded up to the register allocation unit size:

$$R_B = \lceil R'_B, \mathrm{DEF}(R_{alloc\_unit}) \rceil \, . \tag{3.8}$$

Based on resources allocation per threadblock, we can evaluate how these values can limit the maximum number of active threadblocks per SM. The maximum number of threadblocks per SM limited by the maximum number of warps per SM is

$$(B_{SM})_{warp\_limited} = \mathrm{MIN}\left(\mathrm{DEF}(B_{SM\_max}), \left\lfloor \frac{\mathrm{DEF}(W_{SM\_max})}{W_B} \right\rfloor\right) \, , \tag{3.9}$$

where B_{SM_max} and W_{SM_max} are respectively the maximum number of threadblocks per SM and of warps per SM. Similarly, the maximum number of threadblocks per SM limited by registers and shared memory per SM are

$$(B_{SM})_{reg\_limited} = \begin{cases} \left\lfloor \frac{\mathrm{DEF}(R_{SM})}{R_B} \right\rfloor & \text{if } R_T > 0, \\ \mathrm{DEF}(B_{SM\_max}) & \text{otherwise,} \end{cases} \tag{3.10}$$

and

$$(B_{SM})_{Sh\_limited} = \begin{cases} \left\lfloor \frac{\mathrm{DEF}(Sh_{SM})}{Sh_B} \right\rfloor & \text{if } Sh_{B\_req} > 0, \\ \mathrm{DEF}(B_{SM\_max}) & \text{otherwise.} \end{cases} \tag{3.11}$$

At this point, we can calculate the number of active threadblocks per SM as

$$B_{active} = \mathrm{MIN}\left((B_{SM})_{warp\_limited} \, , \quad (B_{SM})_{reg\_limited} \, , \quad (B_{SM})_{Sh\_limited}\right) \, . \tag{3.12}$$

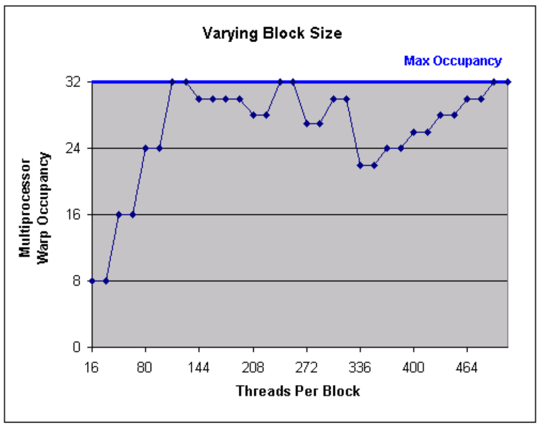
From the latter, it is possible to obtain the number of active threads and warps per SM:

$$T_{active} = T_B \cdot B_{active} \, , \quad W_{active} = W_B \cdot B_{active} \, . \tag{3.13}$$

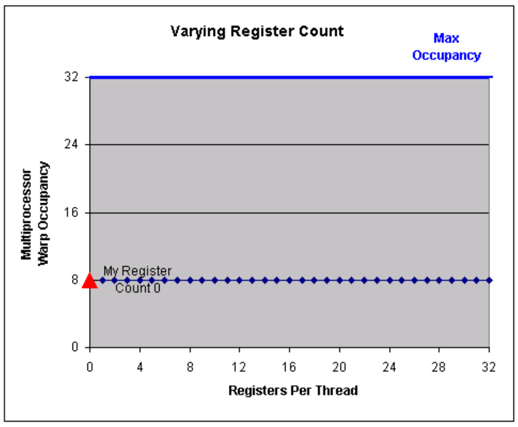
Finally, using W_{active} and DEF(W_{max_active}), we can calculate the multiprocessor occupancy as shown in (3.5).

There is a tool that can help us taking into account all the parameters we described, the CUDA Occupancy Calculator[1]. It is a spreadsheet that automatically evaluates the multiprocessor occupancy once provided the values for the three variables T_B, R_T, and Sh_{B_req}. Furthermore, it can suggest what would happen if we would change those values, showing the multiprocessor warp occupancy curve in three graphs like the ones in Figure 3.8.

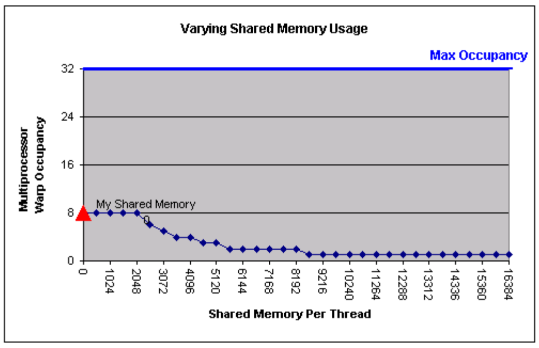
[1] `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`



(a) Dependence on the threadblock size



(b) Dependence on the number of registers



(c) Dependence on the amount of shared memory

Figure 3.8 – Screenshots of multiprocessor warp occupancy graphs from the NVIDIA CUDA Occupancy Calculator using a minimal configuration $T_B = 1$, $R_T = 0$, and $Sh_{B\_req} = 0$.

Synchronization and Consistency

Synchronization is suggested exclusively at thread level. Threadblocks' independence is an important requirements for granting scalability and speed. Also within a threadblock, however, the use of synchronization routines must be carefully controlled and not misused. Mutual exclusions within a threadblock, can be implemented on recent hardware[2] using a combination of barriers and atomic operations, but this goes against the requirement of massive data parallelism required by GPUs to be efficients. Not accidentally, CUDA does not provide any native solution to this kind of approach.
To prevent shared memory access hazards, such as RAW, WAR, and WAW, threads within a threadblock can synchronize through the *__syncthreads()* routine. It forces all the threads to wait for each other. Once all threads have reached the call, execution resumes normally. Host consistency is natively forced for kernels within the same stream, or can be imposed using some synchronization functions. This can be considered a way to force synchronization at threadblock level. A stream is a sequence of operations that execute in order on the GPU. Operations executed on

[2]Atomic operations are just implemented on devices of compute capability 1.1 or higher.

different stream can be interleaved. Stream 0, the default stream, has a special role and acts differently from all the other streams. In particular, every CUDA call on stream 0 blocks until previous calls complete, and no CUDA calls can be overlapped with a stream 0 call. A simple use of streams to overlap a kernel and a memory copy is given in Figure 3.9. All the CUDA instructions issued to a stream can be

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cudaMemcpyAsync(dest, src, size, cudaMemcpyHostToDevice, stream1);
kernelCall<<<gridDim, blockDim, 0, stream2>>>(...);
```

Figure 3.9 – Overlapping computation with memory copy using streams.

synchronized through the call to *cudaStreamSynchronize(stream)*. The function *cudaThreadSynchronize()* can be used to block until all previously issued CUDA calls complete.

Coalesced Access to Non-Cached Memory

Accessing global memory, which is not cached, is an expensive operation. For this reason, it is important to design the most appropriate access pattern. To ensure an efficient data transfer, a first requirement is on the data to be moved. In order to compile an assignment to a single load instruction, the type of the data must be of size 4, 8, or 16 bytes, and the data itself must have address multiple of that size (in other words, the data must be aligned to that number of bytes). CUDA Built-in types [23] are designed to fulfill the requirement. Once data meet type size and alignment requirements, the focus moves to the protocol adopted by CUDA to transfer data from device memory. Again, it depends on the device capability. We focus just on the protocol for devices of compute capability 1.2 or higher. It works for a half-warp, and it can be summarized in the following steps:

1. Find the memory segment that contains the address requested by the lowest numbered active thread. Segment size is 32 bytes for 8-bit data, 64 bytes for 16-bit data, 128 bytes for 32-, 64-, and 128-bit data.

2. Find all other active threads requesting data that lies in the same segment.

3. Reduce the transaction size when only the upper or lower half of the segment is used. This means that the transaction size may shrink to 64 bytes or 32 bytes when initially it was respectively 128 or 64 bytes.

4. Carry out the transaction and mark the serviced threads as inactive.

5. Repeat until all threads in the half-warp are served.

Figure 3.10 shows an example of coalesced access to global memory. When accessing linear memory with a 2D access pattern, the best way to coalesce memory accesses for all half-warps is to have rows with a width that is multiple of half the warp size. Often, it can happen that this requirement is not naturally fulfilled by real data.

For this reason, CUDA provides the function *cudaMallocPitch()*, which allocates global memory padding rows so to satisfy the width constraint. It then returns the address, like every malloc routine, together with a stride that must be used to access correctly the array in global memory. A complete description of pitch-based functions is given in the official reference manual [24].

Also local memory is not cached, and thus accessing it is as expensive as accessing global memory. However, since local memory contains per-thread data, every access would automatically result coalesced.

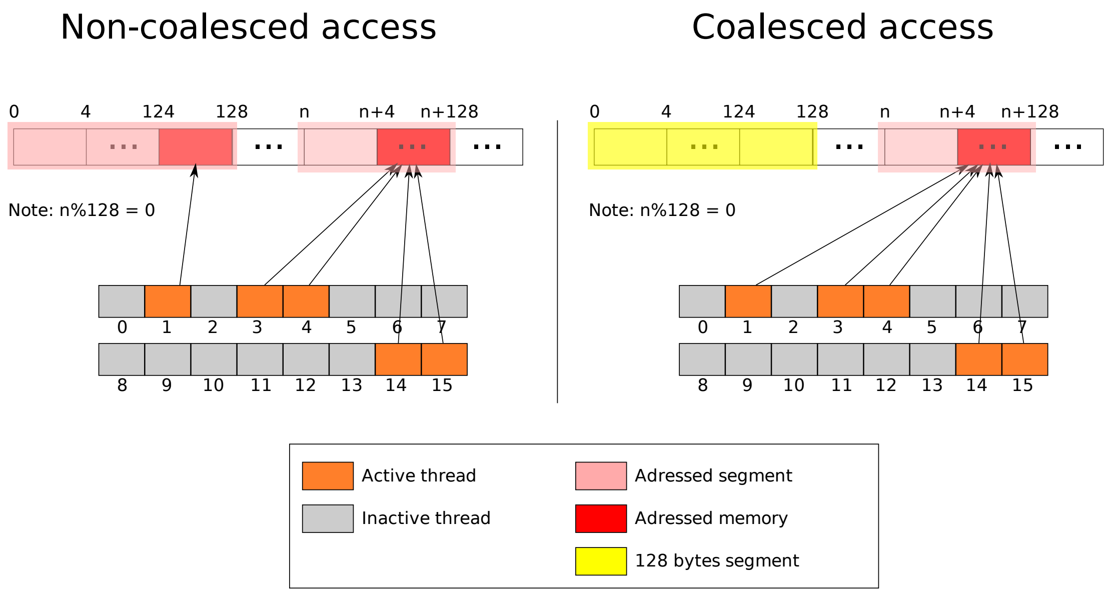


Figure 3.10 – Active threads in the same half-warp making coalesced access to 32 bit words in global memory.

Cached Memory Access

In Chapter 2, we described device memory as split in several areas, among which global and local memory that we have just considered. The other two memory areas that are allocated on device memory are constant and texture memory. The common characteristic is that both spaces are cached. Cache can help to speed up reading instructions. Constant and texture caches are differently implemented, and each supports a specific access pattern. Constant cache is best exploited when all the threads of a half-warp read from the same address. Otherwise, the cost of a read scales linearly with the number of different requested addresses.

Texture memory can be used to take advantage of eventual data locality. Once linear memory or a CUDA array have been allocated, it is possible to associate them to a texture reference in order to fetch them through the texture unit. Textures are cached with a 2D space locality optimization, so they do not require specific access pattern to exhibit good performance. There is an important caveat against texturing from linear memory. Potentially, one could texture from linear memory apply some computation and then write back via global write using the same address pointer (see Figure 3.11). However, the value that has been written back, is not guarantied to be accessed by an ulterior texture fetch within the same kernel call. Kernels can safely texture from linear memory data written by a previous kernel call or memory copy.

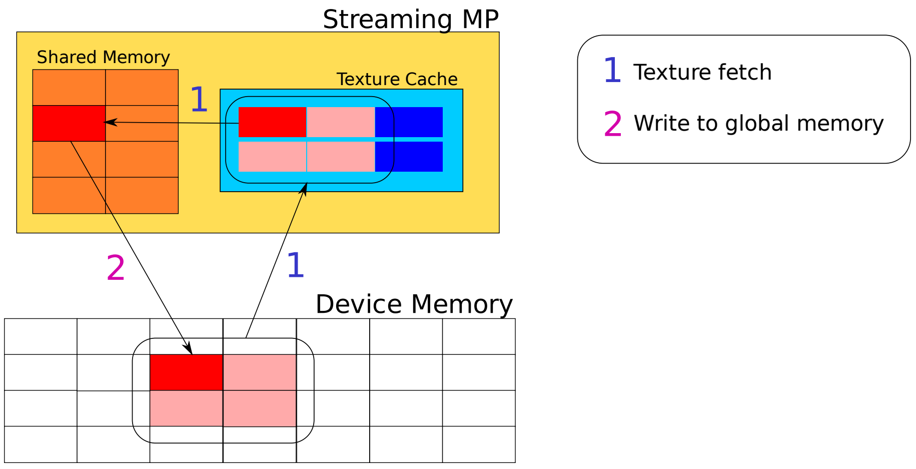


Figure 3.11 – A possible read-write cycle based on linear memory access.

Bank Conflicts

Synchronization is not the only issue for threads belonging to the same threadblock. Performance can also be improved organizing a proper access pattern to shared memory. To allow simultaneous access to shared memory by active threads of a warp, the memory volume is regularly split into different memory modules, called banks. The number of banks is normally equal to the number of threads of a half-warp. A bank conflict can arise when more than one thread in the same half-warp try to access different addresses in the same memory bank. When this happens, the issued loads must be serialized. A memory request with a conflict is split in several conflict-free requests. If a memory request can be served in n conflict-free requests, that request is said to cause n-way bank conflicts. To prevent bank conflicts is then necessary to know how to manage data positioning in shared memory. Banks are organized as shown in Figure 3.12. Successive 32-bit words are stored into successive memory banks. For 32-bit data with 16 banks, a simple access pattern would be to use an odd stride between consecutive addresses[3]. When using data with an arbitrary size of k bytes, it is important to have strides aligned to 4 bytes to prevent conflicts. Figure 3.13 gives two simple examples of conflict-free and 2-way bank conflict shared memory access.

3.2.2 Inter-GPU Communication

Communication between two GPUs, here GPU-0 and GPU-1, can be modeled as a process composed by at least three phases:

[3]Note that, with an even number of banks, an even stride would always involve the same set of banks.

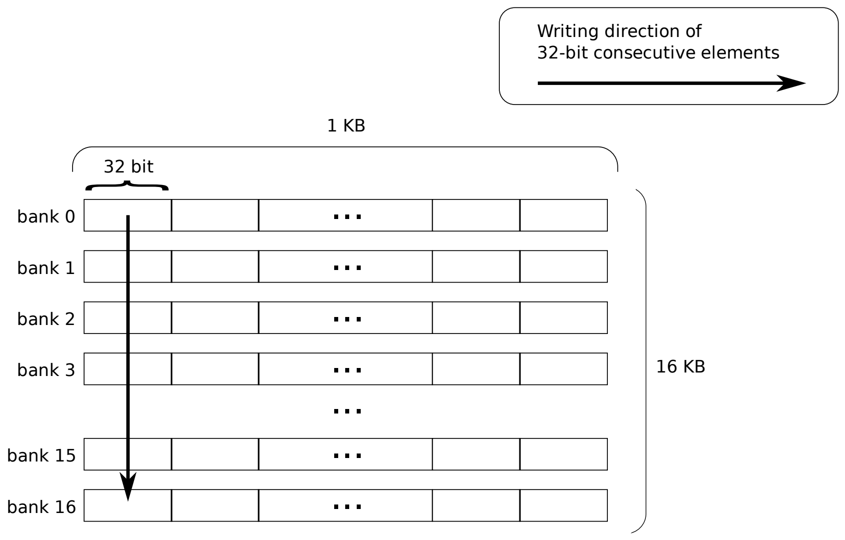


Figure 3.12 – Shared memory structure.

1. Communication between GPU-0 and the host;
2. Communication host-to-host;
3. Communication between the host and GPU-1.

The model is depicted in Figure 3.14. If we suppose that both the GPUs have an identical physical connection to the host, we can express such a model analytically as

$$T_{GPU-GPU} = T_{GPU-host} + T_{host-host} + T_{host-GPU} = 2 \cdot T_{GPU-host} + T_{host-host} \, . \quad (3.14)$$

In Section 3.1.2, we described the communication time in a message-passing multicomputer context. Similarly to a multicomputer, we suppose to express the host-GPU communication time as the linear combination of a certain startup time and the time required to transfer a certain amount of data, leading to the expression

$$T_{GPU-GPU}(n,g) = \sum_i \left(2k^i(n,g)(t_{startup} + w^i(n,g)t_{word}) + T^i_{host-host}(n,g)\right) \, . \quad (3.15)$$

Here n and g are respectively the problem and the system size (in terms of GPUs) The host-to-host communication time $T_{host-host}$, summarizes the time required by all the data movements and synchronization mechanisms part of the communication flow between two GPUs.
The transfer bandwidth between host and device must always be carefully considered. Compacting many small transfers into a big one is often more efficient than performing them separately. Once data are loaded in device memory, it is possible to create intermediate data structures to carry out the computation. A way to increase the bandwidth is to use page-locked memory. This would prevent paging mechanisms from being used on those memory spaces where data have been allocated. Page-locked memory, however, must be used with care. Reducing memory resources may produce system slowdown side effects. Pinned memory introduces a

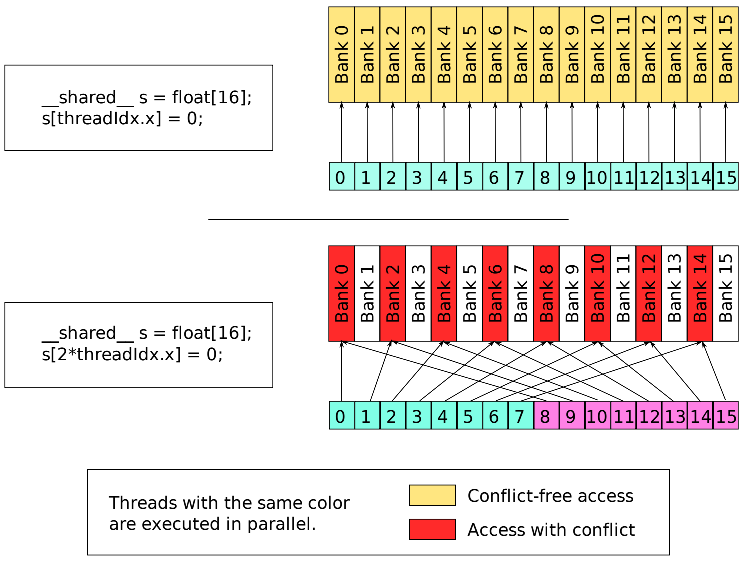


Figure 3.13 – Shared memory access with and without conflicts. Using an odd stride grants conflict free access to the memory.

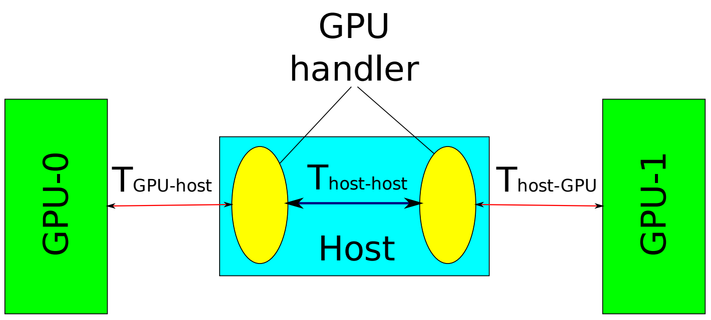


Figure 3.14 – Inter-GPU communication times.

higher startup time relevant for small transfers [27].

Also the $T_{host-host}$ term should be minimized. This can be done exploiting the real parallelism exposed by modern multicore processors. As reported in [17], the use of MPI to communicate between processes on the same node can result in improvable communication overhead. The paper mentions that 2/3 of the communication is spent in buffered MPI_Sendrecv. MPI libraries usually are based on inter-process communication. Using threads that share the same address space could turn out more beneficial. Thus, as an alternative to the message-passing approach, multi-threading with an appropriate synchronization can be used to implement on-node communication among threads associated to different GPU contexts.

Chapter 4

Case Study: PDEs

Partial differential equations (PDE) are very important models required by many engineering applications today. Unfortunately, like in many other cases in mathematics, it is not always possible to find an exact solution to a problem, and even if such a solution solution exists, the way to obtain it could turn out rather unpractical. For this reason, numerical methods implemented on computers are of vital importance to step up engineering applications.

It is not a rare case that solving problems based on PDEs involves very large domains. Just as an example, we could mention weather forecasting and macroeconomic simulations. These an many other examples are sources of enormous storage and computation requirements.

For the sake of our work, we focus on the stencil-based solution of Dirichlet problems [7] using a red-black, successive overrelaxation (SOR) method. In order to understand the development of the next chapters, we recall all the essential elements related to Dirichlet problems and the SOR method. A broader treatment of PDEs and their numerical solution can be found among others in [13, 5, 11, 9].

4.1 PDE Problems

A PDE is a differential equation with partial derivatives of an unknown function of more than one independent variable. Some of the most important practical applications are of second order, where the order is determined by the highest-order partial derivative in the PDE. A generic second-order linear PDE in two dimensions can be represented as

$$a(x,y)u_{xx} + b(x,y)u_{xy} + c(x,y)u_{yy} + d(u_x, u_y, u, x, y) = 0. \tag{4.1}$$

Here, $u(x,y)$ is a twice-differentiable unknown function, a, b, and c are general functions of the independent variables x and y, and d is a linear, (in general) nonhomogeneous function of lower order. As mentioned in [9], PDEs of this form are classified at a point (x,y) according to the sign of the discriminant $\Delta = b^2 - 4ac$ of their characteristic equations, generating three families whose names derive from the analogous conic sections:

- Parabolic, when $\Delta = 0$;
- Elliptic, when $\Delta < 0$;
- Hyperbolic, when $\Delta > 0$.

Normally, the three families are associated to general prototype equations listed in Table 4.1. Any equation of the form (4.1) can be transformed into one of these prototypes by variable changing. It is not goal of this work to describe in details the nature of PDE problems. However, having a clear understanding of the difference between these families, yields to select the right numerical method to solve them. A physical interpretation of the classification of a PDE can help in this sense. Hyperbolic PDEs describe time-dependent, conservative physical processes. This means that the process has a purely oscillatory solution not evolving toward a steady state. Parabolic PDEs describe time-dependent, dissipative physical processes. In other words, parabolic problems present decaying solutions, evolving toward a steady state. Finally, elliptic PDEs describe time-independent systems that already reached their equilibrium.

Table 4.1 – General prototype PDEs.

PDE Family	**Prototype**	**Equation**
Parabolic	Heat equation	$u_t = u_{xx}$
Hyperbolic	Wave equation	$u_{tt} = u_{xx}$
Parabolic	Poisson Equation	$u_{xx} + u_{xx} = f(x, y)$

In order to produce a numerical solution, the mathematical model has to assume a more concrete appearance. What is continuum and exact, must be discretized and approximated. They way discretization and approximations are carried out depends on the specific method. In general, the time-dependency is a relevant characteristic to take into account when trying to numerically solve a PDE. Conservative and dissipative processes must be evaluated step-by-step, starting from given initial values. The way time and space are discretized is quite an important matter. A wrong discretization can lead to inconsistent and unstable results. Heath [9] introduces to semidiscrete and fully discrete methods for time-dependent PDEs, defining and reasoning about consistency and stability properties. On the other hand, domains of steady systems are not limited by time constraints. Each domain's point depends on all its boundary values. This allows to compute the approximate solution to the equilibrium everywhere simultaneously. A possible method to solve such kind of problems is the finite difference method, which is introduced in the next section applied to the prototypical case of the Laplace equation.

4.1.1 Elliptic PDEs: the Laplace Equation

A two-dimensional Poisson equation, can be defined as

$$\nabla^2 u(x, y) = u_{xx} + u_{yy} = f(x, y), \quad u, f \in \mathbb{R},\ (x, y) \in D \subset \mathbb{R}^2 . \tag{4.2}$$

When $f(x, y) = 0, \forall(x, y) \in D$, the (4.2) is called Laplace equation. Solutions to the Laplace equation are often referred as harmonic or potential functions. Elliptic problems are characterized by boundary conditions that a solution must satisfy. Three important boundary value problems are:

Dirichlet problem characterized by the so called essential conditions, where u is prescribed on the boundary curve δD:

$$u(x, y) = \beta(x, y), \quad \forall(x, y) \in \delta D\,. \tag{4.3}$$

Neumann problem characterized by the so called natural conditions, where the normal derivative $u_n = \delta u/\delta n$ is prescribed on δD:

$$u_n(x, y) = \nu(x, y), \quad \forall(x, y) \in \delta D\,. \tag{4.4}$$

Mixed problem characterized by the Cauchy boundary conditions, that can be seen as an imposition of both Dirichlet and Neumann conditions.

In our case, we will take into account the following Dirichlet problem:

$$\begin{aligned} u_{xx} + u_{yy} = 0, &\quad (x, y) \in D = [0, 1] \times [0, 1] \\ u(x, y) = \beta(x, y), &\quad \forall(x, y) \in \delta D,\ 0 \le \beta(x, y) \le B_{max}\,. \end{aligned} \tag{4.5}$$

How to solve elliptic problems is well described in several engineering books, such as [13]. Our interest instead, is in finding a numerical solution to the (4.5). In order to do that, the first step is to define a two-dimensional mesh from the initial domain D, as in Figure 4.1. Mesh points are defined as $x_i = y_i = ih, i = 0, 1, ..., n$, where the mesh size is $h = 1/n$. The second step is to approximate the partial derivatives

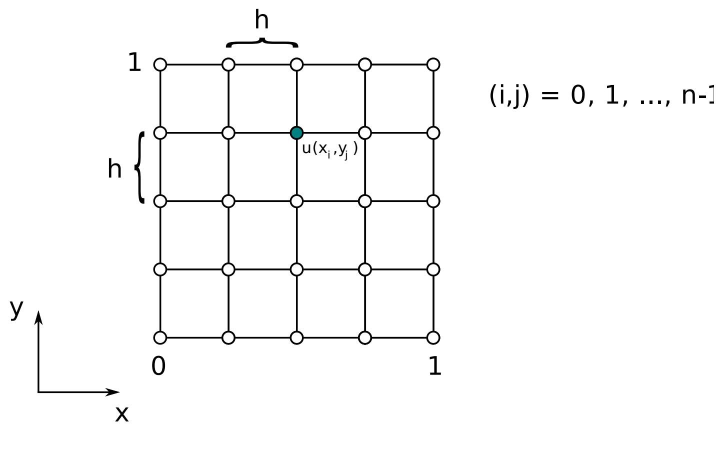


Figure 4.1 – Mesh grid for problem (4.5).

by corresponding difference quotients. An approximation of function $u(x, y)$ can be obtained by recalling its Taylor formula,

$$u(x, y) = \sum_{i=0}^{\infty} \left\{ \frac{1}{i!} \left[(x - x_0)\frac{\delta}{\delta\tau_1} + (y - y_0)\frac{\delta}{\delta\tau_2} \right]^i f(\tau_1, \tau_2) \right\}_{\tau_1 = x_0, \tau_2 = y_0} . \tag{4.6}$$

Alternatively, if we call the differences between the two variables with their centers respectively $\Delta x = x - x_0$ and $\Delta y = y - y_0$, we can rewrite the (4.6) as

$$u(x_0 + \Delta x, y_0 + \Delta y) = \sum_{i=0}^{\infty} \left\{ \frac{1}{i!} \left[\Delta x \frac{\delta}{\delta \tau_1} + \Delta y \frac{\delta}{\delta \tau_2} \right]^i f(\tau_1, \tau_2) \right\}_{\tau_1 = x_0, \tau_2 = y_0} . \tag{4.7}$$

Now, if we develop the above formula taking $x_0 = x, \Delta x = \pm l$ and $y_0 = y, \Delta y = 0$, we obtain

$$u(x + l, y) = u(x, y) + l u_x(x, y) + l^2 u_{xx}(x, y) + o(l^2), \tag{4.8}$$

$$u(x - l, y) = u(x, y) - l u_x(x, y) + l^2 u_{xx}(x, y) + o(l^2) . \tag{4.9}$$

If we add (4.8) and (4.9) neglecting terms in $o(l^2)$, we can derive a central approximation of $u_{xx}(x, y)$,

$$u_{xx}(x, y) \approx \frac{1}{l^2} [u(x + l, y) - 2u(x, y) + u(x - l, y)] . \tag{4.10}$$

A similar result can be calculated for $u_{yy}(x, y)$,

$$u_{yy}(x, y) \approx \frac{1}{k^2} [u(x, y + k) - 2u(x, y) + u(x, y - k)] . \tag{4.11}$$

At this point, recalling that we decided to use mesh size h for both the spatial dimensions, we can rewrite the Dirichlet problem (4.5) in a simpler fashion. Using the notation $u_{i,j}$ to represent the value $u(x_i, y_j)$, we can write

$$\begin{aligned} u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} = 0, &\quad (i, j) \in \bar{D} = [0, 1] \times [0, 1] \subset \mathbb{N}^2 \\ u_{i,j} = \beta_{i,j}, &\quad \forall (i, j) \in \delta \bar{D},\ 0 \leq \beta_{i,j} \leq B_{max} . \end{aligned} \tag{4.12}$$

This finite difference approach uses an approximation of order $\mathcal{O}(h^2)$ [1] with a 5-points coefficient scheme or stencil (Figure 4.2).

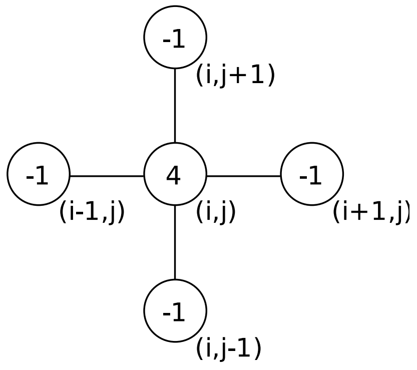


Figure 4.2 – 5-points stencil for the Laplace equation.

[1] For $h \to 0$, the error committed in replacing $\nabla^2 u$ by its finite difference approximation goes to zero as rapidly as h^2 does.

4.2 Iterative Methods

If we apply the stencil to every internal point of the mesh $\bar{D}$, after some reordering we get the system of linear algebraic equations in $m = (n-2)^2$ unknowns

$$\mathbf{A}_{m\times m}\mathbf{u}_m = \mathbf{b}_m \ . \tag{4.13}$$

When m is reasonably "small", many tools exist that can help to solve the system with an effective time complexity. For example, using the Fourier analysis/cyclic reduction (FACR) method, a solution can be found in $\mathcal{O}(m \log\log m) = \mathcal{O}(n^2 \log\log n)$ operations [9]. However, when dealing with large values of m any direct method would require an enormous amount of space to store the matrix $\mathbf{A}$, that has space complexity $\mathcal{O}(n^4)$. As an example, if we decide to use $n = 1000$ floating domain samples per spatial direction (very small compared to real applications), we need a storage capable of containing 10^{12} values (i.e. 4 TB of data).

At this point, direct methods loose appeal, and different methods should be adopted. A possible alternative, is to notice that matrices coming from a finite difference scheme are sparse, and therefore more efficient data structures may be used to store the values, lowering the space complexity to less than $\mathcal{O}(n^4)$ [3]. However, space and work requirements can still be unacceptable for very large matrices. In this cases, iterative methods come as a valid alternative to direct methods. They produce an estimation for the solution starting from a feasible initial one $\mathbf{u}^{(0)}$ successively improved. Theoretically, the convergence to the exact solution is reached after an infinite number of iterations. In practice, one can terminate the computation once some norm of the residual is as small as desired.
We now describe three important stationary iterative methods: Jacobi method, Gauss-Seidel method, and successive overrelaxation method. All of them apply the 5-points stencil to the regular mesh of points $\mathbf{u}^{(k)}$ so to compute the next step's solution $\mathbf{u}^{(k+1)}$.

4.2.1 Jacobi Method

The Jacobi method compute the new approximate solution as the average of the previous solution components at the four neighboring grid points,

$$u_{i,j}^{(k+1)} = \frac{u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)}}{4} \ . \tag{4.14}$$

From a computational perspective, the Jacobi method requires double storage space, since every point of the mesh requires its neighbor not to be updated. Convergence, using the 5-points stencil as described, is always guaranteed from every $\mathbf{u}^{(0)}$. If executed in a sequential fashion, the convergence rate is often too slow. Parallel computers can produce faster results if we consider that every unknown can be calculated simultaneously.

4.2.2 Gauss-Seidel Method

The convergence of the Jacobi method can be improved relaxing its definition. Let us suppose to move in the grid from left to right and from top to bottom. Always considering the 5-points stencil approximation in (4.12), if we use the most recent value of every grid point, we obtain

$$u_{i,j}^{(k+1)} = \frac{u_{i-1,j}^{(k+1)} + u_{i+1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i,j+1}^{(k+1)}}{4} , \tag{4.15}$$

which is known as the Gauss-Seidel relaxation. Again, as for the Jacobi method, convergence is always guaranteed from every $\mathbf{u}^{(0)}$ using the above formulation. from a parallel point of view however, even though the convergence can be reached more quickly with the Gauss-Seidel method. it introduces some sort of sequentiality (i.e. the fact that unknowns must be calculated in a given order) that can impact when designing a parallel program.

4.2.3 Successive Overrelaxation Method

Even greater speed up can be achieved overrelaxating the Gauss-Seidel method. The successive overrelaxation (SOR) method adds to the current approximation $\mathbf{u}^{(k)}$, the increment that we would obtain using Gauss-Seidel relaxed by a parameter ω. Calling the successive approximate solution given by the Gauss-Seidel method $\mathbf{u}_{GS}^{(k+1)}$, we can write

$$u_{i,j}^{(k+1)} = u_{i,j}^{(k)} + \omega \left(u_{GS}^{(k+1)} - u_{i,j}^{(k)} \right) . \tag{4.16}$$

The relaxation parameter ω is a fixed parameter used to either accelerate or decelerate convergence. Acceleration is obtained over-relaxing by a value $\omega > 1$, whereas deceleration under-relaxing by a value $\omega < 1$. Using $\omega = 1$ we obtain the very same Gauss-Seidel formulation. Using the (4.16), SOR is guaranteed to converge for any $\mathbf{u}^{(0)}$ if and only if $0 < \omega < 2$ [31]. As proved in [11], for a problem like the one in (4.5) the value of the optimal relaxation parameter can be calculated as

$$\omega_{opt} = \frac{2}{1 + \sin(\frac{\pi}{n+1})} . \tag{4.17}$$

A more general attempt to formulate ω_{opt} for the SOR method applied to the Poisson equation in n-dimensions can be found in [30]. Even though the convergence speed is quite faster compared to the Jacobi and Gauss-Seidel methods, SOR inherits the sequential sweeping pattern from The Gauss-Seidel definition.

4.2.4 Red-Black Successive Overrelaxation

Sweeping the grid points from left to right and from the top to the bottom introduces a strict sequential dependence between the points. Such a computational pattern can just slowdown a parallel implementation of the fast SOR method. for this purpose, a better ordering should be adopted, in order to expose a higher degree of parallelism. A possible approach, as suggested in [18, 29], is red-black ordering. Using a red-black ordering, grid points are partitioned into red and black points, like in Figure 4.3. In this way, every red point has four black neighbors and every black point four red neighbors. If we call red all those points that have even position (i.e.

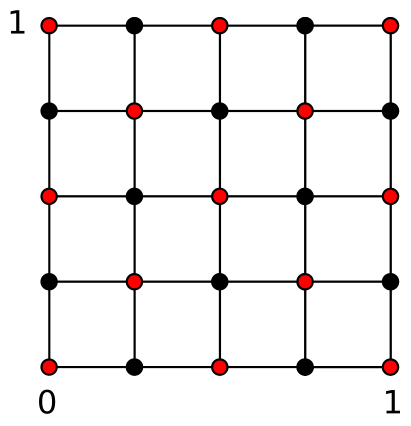


Figure 4.3 – Red-black ordering of the grid points of figure 4.1.

u_{ij} t.c. $(i+j) \mod 2 = 0$), and black all the others, we can reduce the problem of computing one SOR step to the execution of two substeps,

substep 1

In parallel: $$u_{red}^{(k+1)} = u_{red}^{(k)} + \omega \left(\frac{u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)}}{4} - u_{red}^{(k)} \right) \tag{4.18}$$

substep 2

In parallel: $$u_{black}^{(k+1)} = u_{black}^{(k)} + \omega \left(u_{GS}^{(k+1)} - u_{black}^{(k)} \right) .$$

The approximation in every substep can be carry out in parallel for every point belonging to the respective color partition. If we would have a large number of processors close enough to half of the grid points, then a complete SOR step could be computed in constant time using red-black ordering. Such a theoretical assumption, however, is not that far from reality in a multithreaded environment like a GPU.

Chapter 5

Benchmark Application

In this chapter we report about the design and implementation of a benchmark application used to investigate performance factors of a multi-GPU system. The benchmark application will be based on the case study presented in Chapter 4, and the analysis done in Chapter 3. We first summarize our specific benchmarking requirements. Following, we describe the most relevant design and implementation decisions. Details are provided in a top-down fashion, starting from a high-level perspective down to more specific parts of the application.

5.1 Benchmarking Overall Perspective

The target of our work is to analyze how the factors discussed in Chapter 3 can affect performance when working with applications that requires to exchange borders in a multi-GPU system. For this reason, our benchmark application must grant enough freedom in setting the variables related to such factors.

Undoubtedly, the performance of the class of applications taken into account depends on the data domain. The size of the data grid impacts on the amount of work to do and on the amount of data to exchange. Beside that, also the shape of the grid may assume importance. Consider the two grids in Figure 5.1. They both

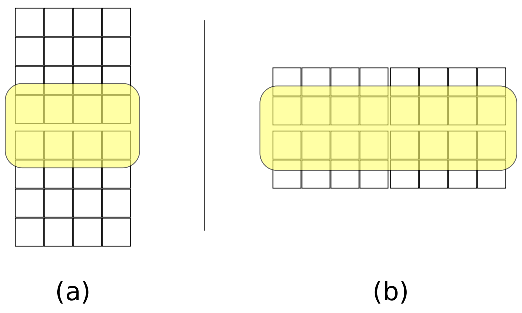


Figure 5.1 – Domains with equivalent area but different shape.

present the same amount of data to process. But they cannot be defined equivalent in terms of processing time. Assuming that it is divided into two separate subgrids to be processed in parallel, and that the borders to exchange have height one, the computation to do on grid 5.1(b) is twice the computation on 5.1(a), whereas the communication in 5.1(b) is half the communication in 5.1(a). Depending on the time required by the two macro-operations, the whole processing time may result different in the two cases.

Another factor is the number of GPUs we want to involve in the calculation. A higher number of processors can lower the computation down, but at the same time increase the communication complexity. Computation and communication themselves are boxes that one can fill up in many different ways, so influencing the final results. In the previous chapter, we decided to implement a PDE solver as a testing application. Of course, the goal of a PDE solver is to actually solve a PDE. Normally, the way an iterative solver reaches its goal is monitoring the approximation of the solution. When some convergence criterion is fulfilled the application can stop its job. This could bestow a further degree of complexity on the parallel application. In a parallel context, fulfillment of common criterion means adding some sort of extra synchronization. Every processing element has to communicate its response to a "referee" that is in charge of assessing the whole situation. Then eventually, the referee node imposes the processing termination.

However, our goal is not to implement a parallel PDE solver in itself. It is just one application among many others that requires border exchange, and , for this reason, is suited for our benchmark. As a consequence, we do not need to define a specific convergence criterion. Better would be for our case to be able to specify how long the calculation has to last. Before describing the benchmark application, we summarize in Table 5.1 those requirements that must be fulfilled in order to provide useful and valid results to the analysis stage. A high level perspective of the benchmark application can be obtained from Figure 5.2. It gets in input the data

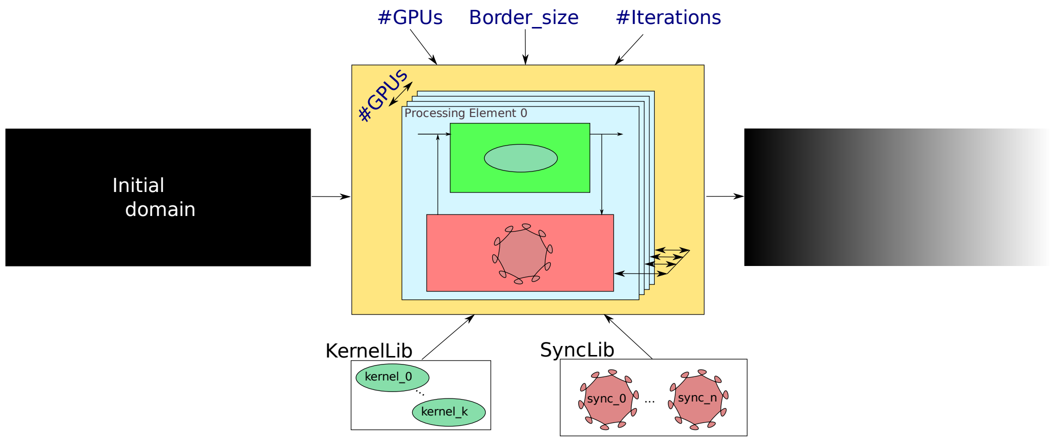


Figure 5.2 – High-level perspective of the benchmark application.

domain to process, the communication and computation method to use, and all the variables discussed above: border size, number of GPUs, and number of iterations.

Every GPU is associated to a different processing context that runs the specified kernel and uses the specified communication method to coordinate and exchange

Table 5.1 – Benchmarking requirements.

Requirement	Description
Domain shape	The application must handle rectangular domains.
Domain size	The application must handle large domains domains able to fit into the available device memory. 16 GB is the amount of device memory provided by the Tesla S1070 when using all the four GPUs.
Stop criterium	The iterative method must be executed for a specified number of iterations rather than being based on some convergence threshold.
Border size	The application must handle variable border sizes.
Number of GPUs	The application must work with all the available GPUs.
Communication fashion	The application must allow the adoption of different communication methods.
Kernel variety	The application must allow the selection of different kernels.
Communication/Kernel combination	The application must indifferently support the possible communication method/kernel combinations.
Performance vs. scalability	Whatever design decision that can allow scalability without degrading performance can be taken into account.

data with the other processing elements. Our application targets Linux-based systems, and is implemented in C++ and C/CUDA. Synchronization approaches are coded at low-level using POSIX threads.

5.2 Exchanging Larger Borders

In his thesis work, Holtet [10] presents a study of stencil-based communication in cluster-like environments. He considered a specific class of applications that requires domain decomposition with border exchange at each iteration. The basic idea behind the described method is that compacting many communications into a single one can reduce the time spent by send/receive routines to set up the communication. However, in order to reduce communication latency, extra-work per node must be introduced. As shown in Figure 5.3, if the border exchanged has width 1, we have to refresh the value of the ghosts elements at each iteration to proceed with the computation. However, if we increase the width of the border up to n elements, this reduces the communication of a factor of n. Of course, as a consequence of this border enlargement, the amount of data to transfer and of computation to perform increase too. After every communication step, the grid of elements can be processed n times, decreasing the width of the border at each computational step.

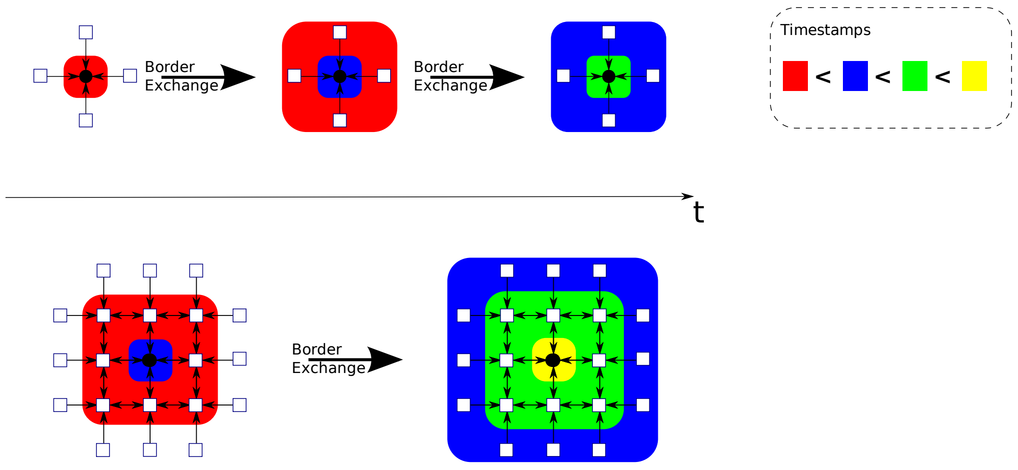


Figure 5.3 – Border exchange for a 5-point stencil with border size 1 and 2. Values with the same color are computed at the same timestamp.

The approach is described as more effective for clusters than supercomputers, showing execution times up to 6X the serial version of the benchmark application. In the conclusion, it is argued that the main reason for that stands in the slower cluster interconnection. Optimal border sizes for clusters were generally larger than those for supercomputers, not showing for the latter any particular performance gain. Our opinion is that what was experienced with supercomputers is more likely what could be experienced with GPUs. The reason is simple. Even though bandwidth is a bottleneck when programming GPUs, the interconnection between host and GPUs is still way faster than an Ethernet connection. The reason why we added the requirement on variable border sizes is to assess to what extent they can be used to handle the bandwidth factor when solving PDEs in a multi-GPU system.

5.3 Data Partitioning

The way data is partitioned can affect communication. Typically, data partitioning for synchronous computation can be carried out splitting the data mesh either in blocks or in strips, generating different communication topologies (Figure 5.4). Wilkinson and Allen [29] argue that, when programming a multicomputer with a reasonable number of processors, strip partitioning is best for communication routines with a large startup time, while block partitioning is best for those with a contained startup time. A reasonable number of processors in their context is $p \geq 9$, which is often satisfied even by small multicomputers. In our case, however, we cannot take this result for granted. We cannot assume such a large number of GPUs connected to the host. On the Tesla S1070, we estimated $t_{startup} \approx 3.3\mu s$ and $t_{word} \approx 15.6\mu s$. Let us suppose to use a squared dataset with n^2 elements and the communication model defined in (3.15) with negligible host-host communication. Using four GPUs, we could decide to split the domain either in blocks or in strips.

Strips are in our case set of rows, so to collect consecutive elements to share among neighbor GPUs. As shown in Figure 5.4, splitting the dataset up in strips,

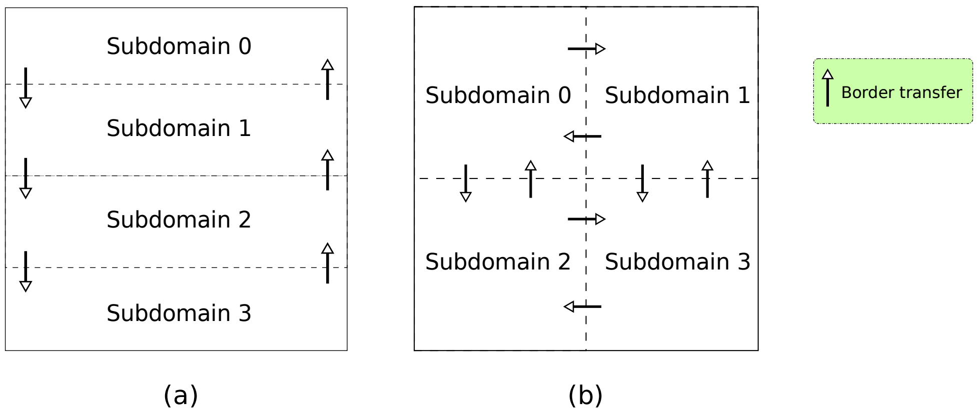


Figure 5.4 – (a) Strip and (b) block partitioning.

requires the two GPUs in the middle to transfer data four times to carry out a correct computation, while the two lateral ones must transfer only twice. Considering in general k layers per border, the transfer times in these two cases are,

$$\begin{aligned} T^{strip}_{lat}(n,4) &= 2T^{strip}_{GPU-GPU}(n,4) = 4k(t_{startup} + nt_{word}) \,, \\ T^{strip}_{mid}(n,4) &= 4T^{strip}_{GPU-GPU}(n,4) = 8k(t_{startup} + nt_{word}) \,. \end{aligned} \tag{5.1}$$

Since, in general, we are not sure that the device uses the same pitch as the host, we suppose the worst case where contiguous rows are sent one after the other (and so the factor k in the formulas above appears outside the parenthesis). When using block partitioning, every GPU has indistinctly two neighbors. Considering vertical and horizontal data transfers separately, we obtain

$$\begin{aligned} T^{block}(n,4) = T^{block}_{ver} + T^{block}_{hor} &= 2T^{block,ver}_{GPU-GPU}(n,4) + 2T^{block,hor}_{GPU-GPU}(n,4) \\ &= 4k\left(t_{startup} + \frac{n}{2}t_{word}\right) + 4\left(\frac{n}{2} + k\right)(t_{startup} + kt_{word}) \,. \end{aligned} \tag{5.2}$$

The four $k \times k$ corners are sent during the horizontal phase, in order to be sure that every block updated its own corner during the vertical tranfer. To understand when block partitioning has larger transfer time, we must solve the two inequalities

$$T^{block}(n,4) > T^{strip}_{lat}(n,4) \,, \tag{5.3}$$

$$T^{block}(n,4) > T^{strip}_{mid}(n,4) \,. \tag{5.4}$$

Working out the (5.3), we find that block partitioning performs slower than strip partitioning when

$$t_{startup} > \frac{-2k^2}{n+2k} t_{word} \,. \tag{5.5}$$

Noticing that the multiplying factor on the right-hand side is always negative, we can conclude that strip-based transfers on the two GPUs associated with the two outer partitions are theoretically always faster than block-based ones. Solving the (5.4), we obtain

$$t_{startup} > \frac{2k(n-k)}{n} t_{word} \,. \tag{5.6}$$

If we consider that n can often be considerably larger than k,

$$t_{startup} > \frac{2k(n-k)}{n} t_{word} \quad \xrightarrow{n \to \infty} \quad t_{startup} > 2k t_{word} \, . \tag{5.7}$$

Differently from the previous result, where two transfers are always faster than four, this time we come to the opposite conclusion. Using the empirical values of $t_{startup}$ and t_{word}, we should conclude that block partitioning is always better than strip partitioning.

So, as a summary, we cannot conclude that one method has better theoretical properties than the other one in our context. Strip partitioning would allow two out of four GPUs to have shorter transfers, as well as block partitioning would improve transfer times of half of the GPUs that deal with two neighbors. Furthermore, we just considered transfer times, while the communication time $T_{GPU-GPU}(n, g)$ generally involves the time required to synchronize communication between the two computing units. This parameter may also influence communication. When using strip partitioning, half of the GPUs not only have half of the data to transfer, but also half of the synchronization to manage, since they both have a single neighbor. Based on these considerations, we finally decide to implement a communication topology based on both strip and block partitioning.

5.3.1 Data Storage

Irrespective of the specific data partitioning technique adopted, the most natural way to store data in main memory for our case study is to map them to consecutive memory locations. In this way, threads can interact with each other using shared memory areas. Borders' addresses within neighbor areas become function of the own address space of a thread and few other parameters (e.g. topological position). As a consequence, the time parameter $T_{host-host}$ expresses the synchronization delay required to orchestrate threads' access to shared memory. Input files are mapped to memory through memory mapping, so to reduce latency due to page swapping during the core computation. Due to our large domain size requirement then, we prefer to exclude the hypothesis of using page-locked memory in our implementation.

5.4 Core Computation

To compute the red-black SOR method as described in Section 4.2.4, we must first decide how to apply red-black ordering coherently to the overall domain. In this way, we grant the convergence of the whole calculation. The decision is to assign red positions starting from element $(1, 1)$. Colors create two disjoint sets of elements. Elements in one set exclusively depends on elements in the other set and themselves. We associate the two substeps in (4.18) to two different CUDA kernels that will be launched one after the other, so to preserve the temporal ordering introduced by the Gauss-Seidel method. Always in a top-down approach, let us now consider the outer host computation.

5.4.1 Host Computation

Every thread is associated with a different GPU and has to elaborate one portion of the whole domain. In Algorithm 5.5 the main computational steps are listed. Right

```
Domain D := mmap("domain.in");

begin in parallel:

  select_device(thread_idx);

  {Load subdomain on device with bs layers per border}
  load_subdomain(D[thread_idx], bs);

  for i := 0 to iterations/bs do
  begin
    for i := 0 to bs do
    begin
      {Red/Black SOR reducing border's size at each iteration}
      RBSOR(D[thread_idx], bs-i);
    end

    {Exchange border with neighbors}
    border_exchange(D[thread_idx], bs);

  end
end in parallel.
```

Figure 5.5 – Overall benchmark algorithm.

after having mapped the input file to main memory, a number of threads are spawn, and different memory areas are assigned to each of them. Different partitioning and synchronization approaches imply different allocations. Section 5.5 describes this point more in detail. The SOR parameter, if not explicitly indicated from command line, is computed using (4.17) in Section 4.2.3.

Every thread is characterized by an index that unambiguously associate it to its area and the GPU it has to use. First operation for every thread is to select a GPU according to its index. The solver must run for a certain number of iterations. After every iteration, to produce a consistent approximation of the solution, borders must be exchanged. Using the approach described in Section 5.2, a certain number of iterations may run locally. In this last case, thicker borders must be exchanged once the computation has consumed all the outer extra layers in the border.

5.4.2 Device Computation

The red and black SOR steps are implemented with two different CUDA kernels. To run a single red-black SOR iteration, we need to invoke red and black kernels one after the other. Since both the kernels use the same data and are run on the default stream 0, calling them in sequence automatically imposes a threads synchronization point in between the two calls. Such a point excludes every chance to start black computation in advance. Before describing the most relevant details of the design and implementation of the kernels, we summarize the most important step in their computational flow. Every kernel must:

1. Load data from global memory;
2. Compute the stencil-based operations;
3. Write data back to global memory.

Execution Configuration Setup

In short, setting up the execution configuration means deciding how to map computation to the thread hierarchy and how to split, and eventually share, resources. Mapping computation to the thread hierarchy comes also very natural thanks to the streaming processing model of the GPU. Since every red (black) point is independent from all the other red (black) elements, assigning the computation of one domain point to a single thread supports the natural parallelism of the task. So the 2D domain can be mapped to a 2D grid, where several threadblocks carry out the red/black computation. The number of threads per threadblock and the amount of resources at their disposal, is strictly related to the goal of taking advantage of the hardware as much as possible. In Section 3.2.1, we introduced multiprocessor warp occupancy, SM_{occ}, as a reference parameter to estimate if a certain execution configuration support or not an efficient repartition of the resources. Maximizing SM_{occ} requires to increase as much as possible the number of active threadblocks per SM:

$$\mathrm{MAX}(\mathrm{SM}_{occ}) = \mathrm{MAX}\left(\frac{\mathrm{W}_{active}}{\mathrm{DEF}(\mathrm{W}_{max\_active})}\right) \Leftrightarrow \mathrm{MAX}(W_{active}) = \mathrm{MAX}(W_B \cdot B_{active}),$$

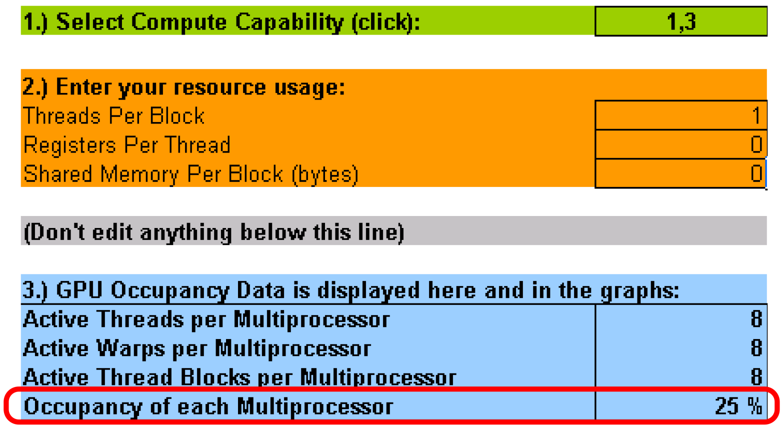
and

$$\mathrm{MAX}(B_{active}) \Leftrightarrow \mathrm{MAX}(\mathrm{MIN}\left((B_{SM})_{warp\_limited}\,,\ (B_{SM})_{reg\_limited}\,,\ (B_{SM})_{Sh\_limited}\right)).$$

The max-min problem shown above presents a large range of feasible solutions. These are obtained tuning the parameters that can potentially limit the maximum number of active threadblocks per SM, i.e. the number of threads per threadblock T_B, the number of registers per thread R_T, and the requested amount of shared memory per threadblock Sh_{B_req}.

We already mentioned the CUDA Occupancy Calculator as a tool to speed up this tuning phase. An initial combination of these factors come from the problem requirements. Our kernels must essentially apply the stencil-based operation to a bunch of points. Stencil-based operations involves neighboring values (four in our case) that are typically not aligned when data are stored linearly. Furthermore, the same value can be required more than once. For these reasons, our solution can really benefit from the use of shared memory in our kernels. The amount of shared memory, however, depends on the number of threads per threadblock, since the more the threads the more the neighbors, and the more the neighbors the more memory must be shared among the threads.

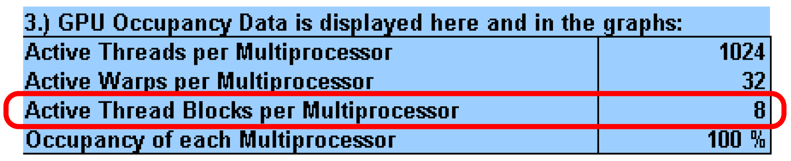
Here is a possible way to tune the parameters based on the assumptions made so far. First thing to do is to set the right compute capability (i.e. 1.3 in our case). Since we want to define the amount of shared memory based on the number of threads, we start assuming a very low number of threads, say $T_B = 1$. The calculator automatically calculates and shows the GPU occupancy parameters, as reported in Figure 5.6. We can see how using a single thread per threadblock would



1.) Select Compute Capability (click):	1,3
2.) Enter your resource usage:	
Threads Per Block	1
Registers Per Thread	0
Shared Memory Per Block (bytes)	0
(Don't edit anything below this line)	
3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	8
Active Warps per Multiprocessor	8
Active Thread Blocks per Multiprocessor	8
Occupancy of each Multiprocessor	25 %

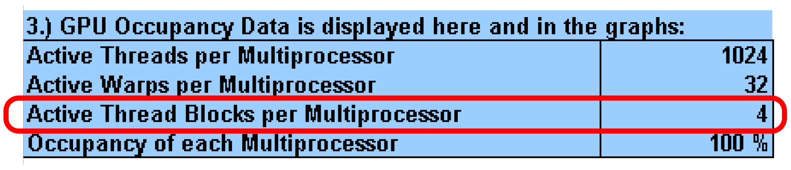
Figure 5.6 – GPU occupancy parameters using a single thread per threadblock.

drastically reduce the occupancy of the SMs. However, the Calculator also presents us the three graphs in Figure 3.8. These graphs give us an idea about how the warp occupancy may change moving the parameters' values. We notice the presence of three peaks in the graph in Figure 3.8(a) that would move the occupancy to its maximum (i.e. $W_{active} = 32$) and so to a 100% occupancy. The difference would be in the number of active threadblocks per SMs. A larger threadblock would require more warps to be processed, consequently reducing the possible active threadblocks per SM (Figure 5.7). Supposing to set a value in between $T_B = 112$ and $T_B = 128$,



3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1024
Active Warps per Multiprocessor	32
Active Thread Blocks per Multiprocessor	8
Occupancy of each Multiprocessor	100 %

(a)

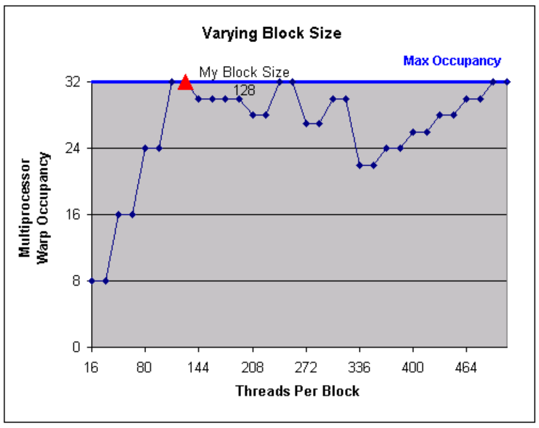


3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1024
Active Warps per Multiprocessor	32
Active Thread Blocks per Multiprocessor	4
Occupancy of each Multiprocessor	100 %

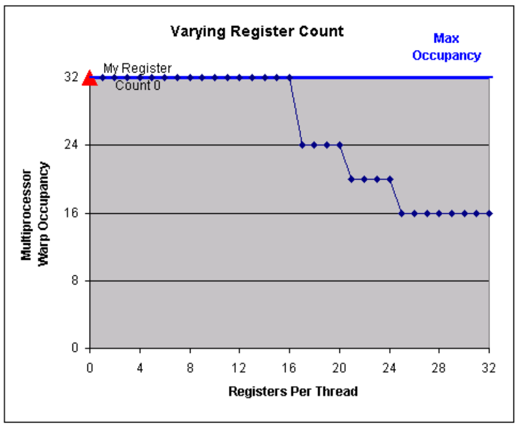
(b)

Figure 5.7 – GPU occupancy parameters with (a) $T_B = 128$ and (b) $T_B = 256$.

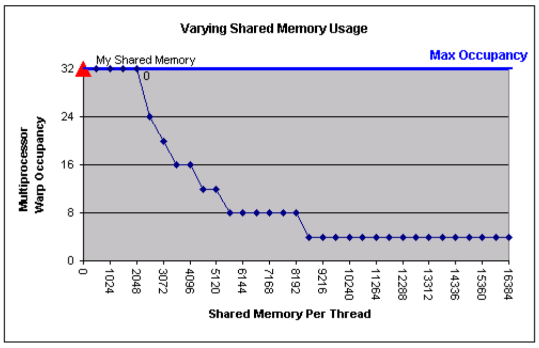
the warp occupancy status is shown in Figure 5.8. With this configuration we have at disposal 2048 KB of shared memory per threadblock and up to 16 registers per thread. This allows us to load in shared memory up to $2048/4 = 512$ floating points. At this point, the next goal is to design the internal structure of each kernel based on this constraints.



(a) Dependence on the threadblock size



(b) Dependence on the number of registers



(c) Dependence on the amount of shared memory

Figure 5.8 – Warp occupancy with $112 \leq T_B \leq 128$.

Red SOR Kernel

Based on the previous arguments, we have the possibility to instantiate between 112 and 128 threads per threadblock, loading at most 512 domain elements. Moreover, would be beneficial to use at most 16 registers per thread. Our solution presents threadblocks as 1D collections of 112 threads working on an area of 4×112 elements. In this way, it is possible to acquire 1792 KB in exactly 28 coalesced accesses (Figure 5.9). Since not always the height of an area is divisible by four, a variable keeps track of whether lines are three or four:

```
int ok3 = gridPosBase_j+3 < h;
```

Of the 2048 KB that could be used without degrading occupancy, 256 bytes are not directly used. Part of them are used to store the kernel's actual parameters together with the execution configuration parameters. Another part, instead, remains completely unused. Adding few values per row, however, would require four more accesses to global memory to transfer less data than what could be transfered.

```
if(gridPosBase_i+i < n)
{
  sgrid[ATS(i,0)] = dgrid[ATD(i,0)];
  sgrid[ATS(i,1)] = dgrid[ATD(i,1)];
  sgrid[ATS(i,2)] = dgrid[ATD(i,2)];
  if(ok3)
    sgrid[ATS(i,3)] = dgrid[ATD(i,3)];
}
```

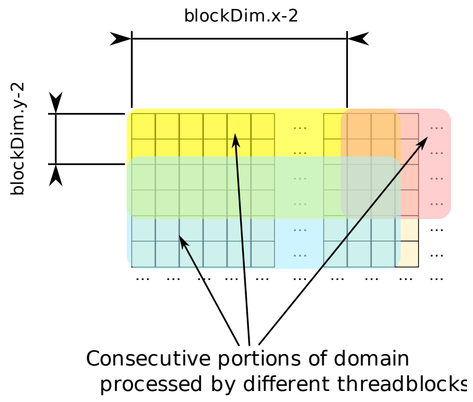
Figure 5.9 – Code to load four lines of memory from global memory.

In terms of performance, that would be worse than wasting few bytes. The values excluded are in any case processed by some other thread within some other threadblock, without introducing irregularities in the way to access global memory. Access to global and shared memory is done through two macros reported in Figure 5.10. They are based on the strides of each specific memory region. The portions of do-

```
#define ATD(x,y) (pitch)*(gridPosBase_j+(y)) + (gridPosBase_i+(x))
#define ATS(x,y) (local_w)*(y) + (x)
```

Figure 5.10 – Indexing macros.

main assigned to contiguous threadblocks overlap on those elements that act as local borders. Since such portions have borders with unitary width, frames must slide 2 elements vertically and $blockDim.x - 2$ elements horizontally. So the position of a threadblock is computed taking this sliding effect into consideration, as shown in Figure 5.11. Data loaded in shared memory are processed by all the threads but


blockDim.x-2
blockDim.y-2
Consecutive portions of domain
processed by different threadblocks

```
int gridPosBase_j = blockIdx.y
    *2;
int gridPosBase_i = blockIdx.x*
    (blockDim.x-2);
```

Figure 5.11 – Code to determine the position of a threadblock in its grid.

two. Threads are mapped reflecting the red access pattern, as shown in Figure 5.12. Every thread in a threadblock is mapped to an element based on the absolute coordinates of the element in the overall domain, as shown in Figure 5.13. Only odd rows of the overall domain have red elements in locations $(1, i)$. Different GPUs, however, enumerate blocks relatively to their local portion of domain. Due to the borders, every second local iteration makes the local domain start with an odd line of the overall domain, thus having in position (1,1) a black spot. To understand whether to start from (1,1) or (2,1), one of the parameters to the kernel contains the

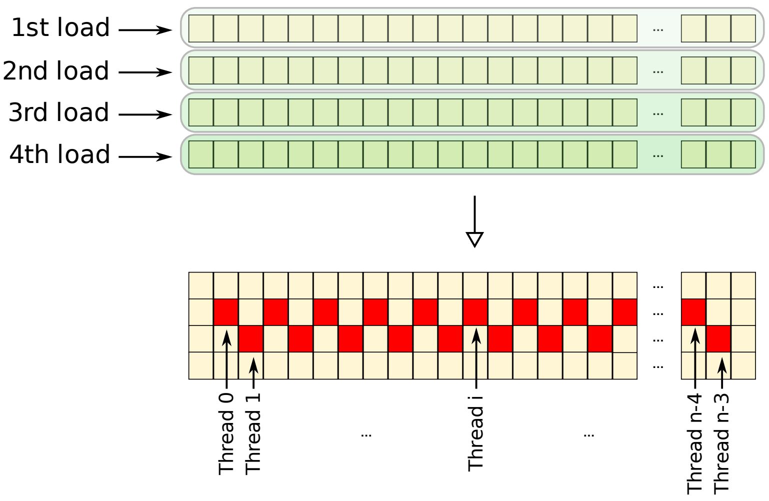


Figure 5.12 – Threadblock mapping to shared memory elements during the first two phases of the red SOR kernel.

```
int r1 = (abs_start+gridPosBase_j
    +1)&1;

int sign = r1 + __mul24(1-r1, -1);

//Red indexes of computation
int com_i = i+1;
int com_j = (2-r1) + __mul24(sign,
    1&i);
```

```
int r1 = (abs_start+gridPosBase_j
    +1) % 2;

//Red indexes of computation
int com_i = i+1;
int com_j = 1;
if (r1)
  com_j = 1 - (i%2);
else
  com_j = 2 + (i%2);
```

Figure 5.13 – Code to determine the red coordinates in shared memory with analogous branch-based version.

absolute *vertical* coordinate of the first element of the local domain. In this way, the kernel is able to understand if point (1,1) in shared memory is red or not, and set the computing y coordinate (i.e. com_j) to the right value.

Code in Figure 5.13 computes indexes using integer arithmetics instead of branching. This is for preventing warps from being split up and serialized. Even filling up the device memory of all the GPUs on the Tesla S1070 with a domain of nearly 16 GB, indexes would never need more than 24 bits to be represented. Thus fast 24-bit integer multiplications are used instead of the slower 32-bit ones (4 cycles per warp against 16).

Once sure that the thread would not operate outside the local and the global margins, the computation of the red SOR step is done accessing the two central shared memory rows (when the last line is valid, otherwise just row one). The kernel applies the stencil-based operations described in (4.18) using the passed overrelaxation parameter (Figure 5.14). Access to shared memory is conflict free. We achieve this storing values in consecutive banks with unitary stride. Since the threadblock size is multiple of half the warp size, a thread with index i always accesses the same memory bank when reading from or writing to elements (i,0), (i,1), (i,2), or (i,3) (see

```
if((i<local_w -2)&&(gridPosBase_i+i<n-2))
{
  if(ok3)
    sgrid[ATS(com_i, com_j)] +=
     omega*(
       (sgrid[ATS(com_i-1, com_j)] + sgrid[ATS(com_i+1, com_j)] +
        sgrid[ATS(com_i, com_j-1)] + sgrid[ATS(com_i, com_j+1)])/4.f - sgrid[ATS(
            com_i, com_j)]
     );
  else if(com_j != 2)
    sgrid[ATS(com_i, com_j)] +=
     omega*(
       (sgrid[ATS(com_i-1, com_j)] + sgrid[ATS(com_i+1, com_j)] +
        sgrid[ATS(com_i, com_j-1)] + sgrid[ATS(com_i, com_j+1)])/4.f - sgrid[ATS(
            com_i, com_j)]
     );
}
```

Figure 5.14 – Red SOR step application.

Figure 5.15). Finally data are written back to global memory through 7 coalesced

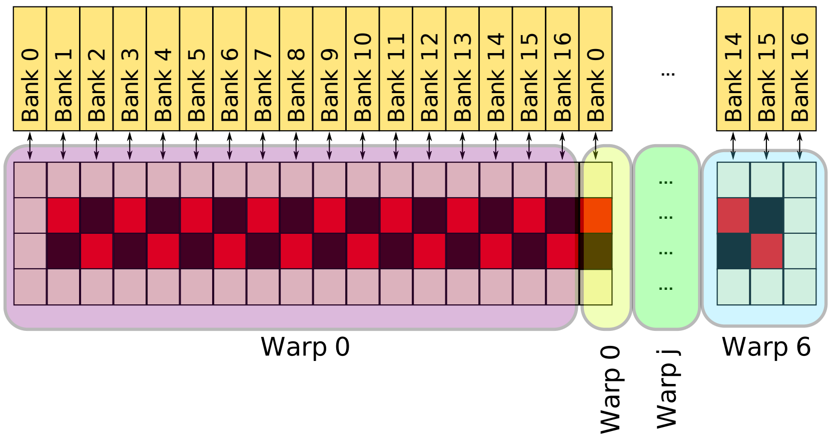


Figure 5.15 – Threadblock access pattern to shared memory.

writes (Figure 5.16). There are two synchronization points throughout the kernel. One is right after the data loading phase, so to apply the SOR step using the right values. The second point is after the computation phase, in order to write all the updated values back to global memory. The complete kernel is entirely reported

```
if((i<local_w -2)&&(gridPosBase_i+i<n-2))
{
  if(ok3)
    dgrid[ATD(com_i, com_j)] = sgrid[ATS(com_i, com_j)];
  else if(com_j == 1)
    dgrid[ATD(com_i, 1)] = sgrid[ATS(com_i, 1)];
}
```

Figure 5.16 – Code to write data back to global memory.

in Appendix B. The number of registers can be monitored compiling with option ”-ptxas”. It produces a list with all the kernels that have been compiled and the main resources required by each of them. As a general property of all the kernels, registers involved are within the range 9-12.

Black SOR Kernel

The black SOR kernel is basically very similar to the red one. What changes is substantially the computation of the indexes of the points where to apply black SOR. Figure 5.17 shows the code used to compute such indexes.

```
int r1 = (abs_start + gridPosBase_j
     +1)&1;

int sign = (1-r1) + __mul24(-1,r1);

//Black indexes of computation
int com_i = i+1;
int com_j = (1+r1) +
  __mul24(sign,1&i);
```

```
int r1 = (abs_start+gridPosBase_j
    +1) % 2;

//Black indexes of computation
int com_i = i+1;
int com_j;
if (r1)
  com_j = 2 - (i%2);
else
  com_j = 1 + (i%2);
```

Figure 5.17 – Code to determine the black coordinates with analogous branch-based version.

Texture-based Approach

Our solution makes use of coalesced accesses to global memory to improve the performance of read/write operations. However, in order to investigate how beneficial can different approaches to the memory turn out, we designed a texture-based version of the kernels. Our access pattern indeed, presents a good degree of locality since memory elements are accessed sequentially.

Every thread is associated to a different texture bound to its own portion of domain. Textures must be bound at each iteration, since some of the outer layers contain ghost elements. Borders containing such layers must be shrunk until the core area is not reached, thus reducing the dimension of the area bound to the texture. The code used to load data from global memory (Figure 5.9) is modified so to fetch elements from texture (Figure 5.18). The proper texture is selected by an

```
if(gridPosBase_i+i < n)
{
  sgrid[ATS(i,0)] = selectTex1D(dev, ATD(i,0));
  sgrid[ATS(i,1)] = selectTex1D(dev, ATD(i,1));
  sgrid[ATS(i,2)] = selectTex1D(dev, ATD(i,2));
  if(ok3)
    sgrid[ATS(i,3)] = selectTex1D(dev, ATD(i,3));
}
```

Figure 5.18 – Data loaded through texture fetching.

inline, device function based on the specific device number, as shown in Figure 5.19. Textures are not writable entities. After the computation has been carried out, data are written back to global memory as described in Figure 5.16.

```
float __device__ selectTex1D(int gpu, int pos)
{
  switch(gpu)
  {
    case 0:
      return tex1Dfetch(texRef_0, pos);
    case 1:
      return tex1Dfetch(texRef_1, pos);
    case 2:
      return tex1Dfetch(texRef_2, pos);
    case 3:
      return tex1Dfetch(texRef_3, pos);
  }
  return 0.f;
}
```

Figure 5.19 – Texture select & fetch function.

Impact of Data Partitioning on Kernels

When the domain is horizontally strip partitioned, every threadblock that starts loading data in shared memory from an even row (with respect to the entire domain) is granted to have red cells in position $(2i+1, 1)$ and $(2i, 2)$. This because the ideal frame that contains the data to load in shared memory slides horizontally by an even number of columns ($blockDim.x - 2 = 110$). When the domain is divided in blocks, some local areas might have the opposite color ordering than the whole domain (i.e. black top-left corner). In this ituation, threadblocks must check a more general condition than the one in Figure 5.13 to understand whether position (1,1) in shared memory is red or not. With respect to the absolute coordinates of the whole domain, red cells lie either in position $(2k, 2k)$ or $(2k+1, 2k+1)$. Thus, the bitwise arithmetic expression to check the color of cell (1,1) in shared memory becomes like the one in Figure 5.20.

```
int r1 = ((abs_start_i + gridPosBase_i +1)&1)&((abs_start_j + gridPosBase_j +1)
    &1);
r1 |= !((abs_start_i + gridPosBase_i +1)&1) & !((abs_start_j + gridPosBase_j +1)
    &1);
```

Figure 5.20 – Code to determine the red coordinates in shared memory when using block partitioning.

5.5 Communication and Synchronization

Communication is the second important part of the code executed by every thread after computing one or more complete SOR iterations on their portion of domain. Aside the kind of subdivision used to split the domain up, there must be a protocol that regulates threads' communication at each border. Let us suppose to look at the frontier between two regions A and B. After the computation process is over at iteration i, the device memories of the GPUs associated with And B contain two updated portions of the starting domain A^i and B^i. To proceed with the next iteration, they first have to exchange borders between each other. This can be done writing and reading just those data that compose the border, and therefore are necessary to continue the computation. However, threads' scheduling is completely unpredictable, and data consistency must be assured. A way to do that is to grant that the following two conditions are always respected:

- A border cannot be written at iteration i if the other thread has not read it yet at iteration $i-1$;
- A border cannot be read at iteration i if the other thread has not written it yet at iteration $i-1$.

In order to implement such a protocol, we associate two counters to each border of the two regions A and B. One counter, r, counts how many times a border has been read, while the other one, w, how many times a border has been updated. If we suppose that they are initialized to zero and that we count iterations starting from zero, then a thread $T0$ has read its neighbor $T1$'s border at iteration i if $r(T1) == i+1$, and it has written to main memory its own border if $w(T0) == i+1$. The algorithm in Figure 5.21 describes the procedure in which a thread updates its border's copy in main memory and acquires the border of the neighbor. Before executing the first

```
procedure borderExchange(Thread T, Neighbor N,
  Region R_T, Region R_N, DeviceRegion D_R, integer iter)
begin
  {Eventually wait until N reads T's border}
  while (r(T) <> iter+1);

  {Update border in main memory with the most recent from D_R}
  R_T.border = D_R.myBorder;

  {T wrote its border once more}
  w(T) := w(T)+1;

  {Eventually wait until N writes its border}
  while (w(N) <> iter+1);

  {Update border in device memory with the most recent from R_N}
  D_R.NeighborBorder = R_N.border;

  {T read N's border once more}
  r(N) := r(N)+1;
end
```

Figure 5.21 – Border exchange algorithm.

SOR computation, each thread has to write to shared memory for the first time and to load data (borders included) in global memory. This increments both the counters for all the borders. After this startup phase the computation and communication follows up. At each iteration, when the computation is over, threads run the illustrated procedure for each frontier they face. The communication topology imposes a structure to the communication which will be analyzed later on in this section. At iteration i, for each of its frontiers, a thread starts waiting for the neighbor to read its border if it has not done it yet. In that case, it means that the neighbor still has to run the i^{th} computation. Once it is allowed, the thread updates its shared border in main memory and increments the writing counter associated with that memory area. Afterwards, it checks that the neighbor has also updated its border in main memory. If it has not the thread put itself in a waiting status, otherwise it loads the updated neighbor's border in device memory for the next $(i+1)^{\text{th}}$ computation and increments the associated read counter.

A formal and concise description of the possible states during the synchronization phase can be obtained from the Petri net[1] in Figure 5.22. In our discussion so far, we

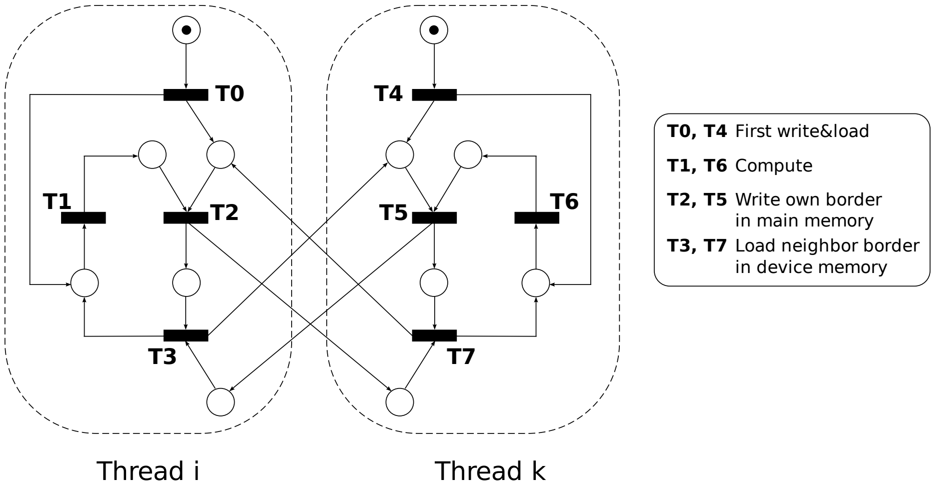

T0
T1
T2
T3
T4
T5
T6
T7
T0, T4 First write&load
T1, T6 Compute
T2, T5 Write own border in main memory
T3, T7 Load neighbor border in device memory
Thread i
Thread k

Figure 5.22 – Petri net describing the interaction between two neighbor threads.

implicitly assumed that threads were operating atomically on shared elements, i.e. borders and counters. In reality, such an atomicity property must be programmed. We associate a mutex and a condition variable to each counter. When a thread wants to check whether the neighbor has already read or not its border, it has to lock the mutex associated to that border before accessing the border's read counter. If then the counter is not up-to-date, the thread waits on the counter's condition variable releasing the mutex. When the counter reaches the expected value, the neighbor signals the condition variable throwing the waiting thread in again. Then when the thread has moved its own border to main memory, it acquires again the mutex on the border so to safely increment the writing counter. Once the counter has been incremented, it signals the condition variable associated to the counter so to notify the eventually waiting neighbor. A similar approach is adopted for the reading part of the exchange border procedure.

5.5.1 Impact of Data Partitioning on Communication

Different data partitioning techniques bring to different communication topologies. We want to analyze the two methods we took into account in this chapter, i.e. strip partitioning and block partitioning.

[1]Carl Adam Petri is a German mathematician and computer scientist. His Petri nets are well suited for modeling the concurrent behavior of distributed systems.

Strip Partitioning Communication

We apply strip partitioning vertically, as shown in Figure 5.4(a). In order to minimize the time spent by threads to wait for each other, the communication direction for each of them is alternated in a chessboard fashion. In this way, those threads in the middle are provided with a clear policy about which direction to consider first. If the thread has an odd rank then it starts working in the northern frontier, otherwise in the the southern one. The process is depicted in Figure 5.23. In principles, we could refine the communication process even further. Considering that border exchanges across different frontiers are totally independent from each other, we could split the job of the interior threads in two. Using a boss-worker model and n GPUs, if an interior thread would assign the task of exchanging data across the two frontiers to two other worker-threads, we would end up with $n-1$ couples of threads carrying out their jobs in parallel (and so potentially at the same time if the CPU would have enough cores at its disposal, see Figure 5.24). However, this

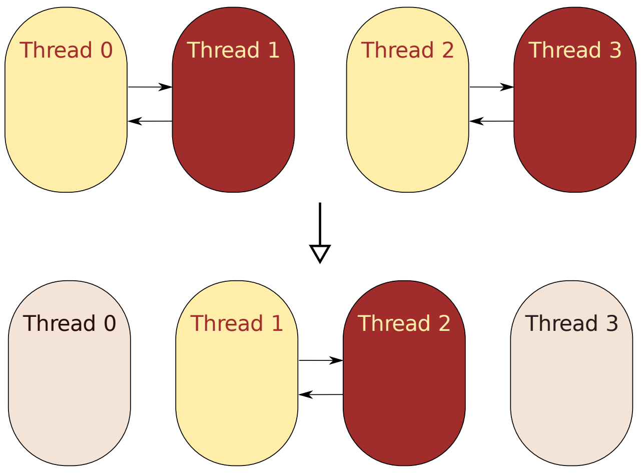


Figure 5.23 – Chessboard-ordered threads communication.

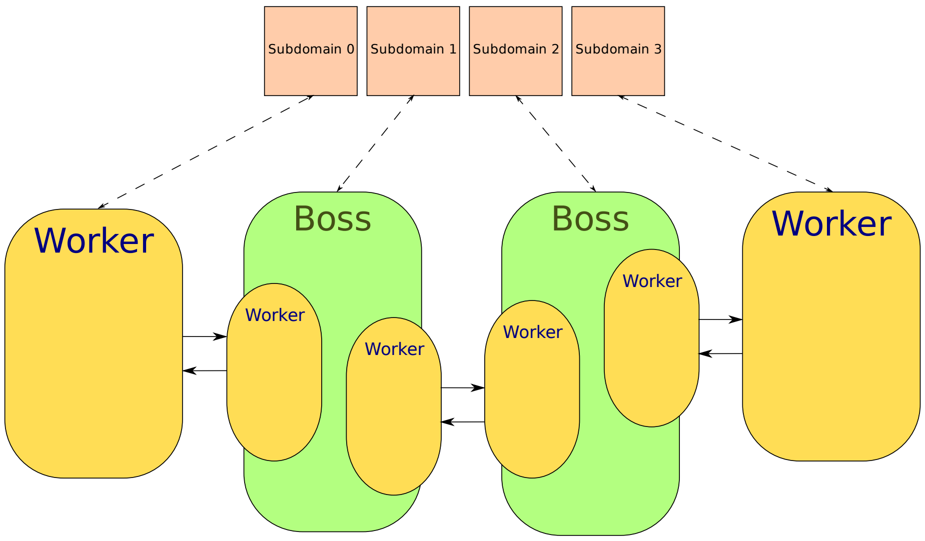


Figure 5.24 – Boss-worker threads communication.

approach is not practicable using the CUDA Runtime API, since it lacks threads' contexts migration capabilities. After a thread is created, it is associated to a new device context where it can manage all that is needed to handle a GPU. If at a certain point that thread would create another new thread, the new one could not refer to the same context as the father, for instance loosing the possibility to share memory on the same GPU. To our knowledge, the only way to implement such a solution is to use the thread migration API, but it is part of the CUDA Driver API, which use would require to rewrite the entire application from the beginning with a different approach. To go back to the chessboard approach, Figure 5.25 shows a sketch of the communication code used when data is strip partitioned.

```
if(rank%2) //if odd start from NORTH otherwise from SOUTH
{
  write_ghosts(rank, north_border);
  read_ghosts(rank-1);
  write_ghosts(rank, south_border);
  read_ghosts(rank+1);
} else
{
  write_ghosts(rank, south_border);
  read_ghosts(rank+1);
  write_ghosts(rank, north_border);
  read_ghosts(rank-1);
}
```

Figure 5.25 – Communication among threads managing a strip partitioned domain.

Block Partitioning Communication

We design our block partitioning supposing that all the GPUs in the Tesla S1070 are involved in the computation. Using just 2 GPUs would bring to a strip partitioned domain. Let us consider the case represented in Figure 5.4(b). Communication in this case must be organized so to correctly transfer those elements that belong to overlapping areas.

To reach this goal, we can require the threads to carry out the communication first vertically and then horizontally, creating pairs of communication flows. When exchanging data in the north-south direction, threads must write their entire borders from device to host, and read just those elements that are exclusively shared by the couple (i.e. excluding the central elements shared by all the four threads).

During the second phase threads communicate on the east-west axis. Here, they first write to main memory their updated borders disregarding those elements already written during the first phase. Afterwards, they read their neighbors' borders including the central elements shared by all the threads. The approach is graphically explained in Figure 5.26. Moving the central points during the last phase grants that the central values have been correctly updated, since they are written during the first group of writes in the north-south communication. An implementation draft for this kind of communication is given in Figure 5.27 for a block with index *rank*.

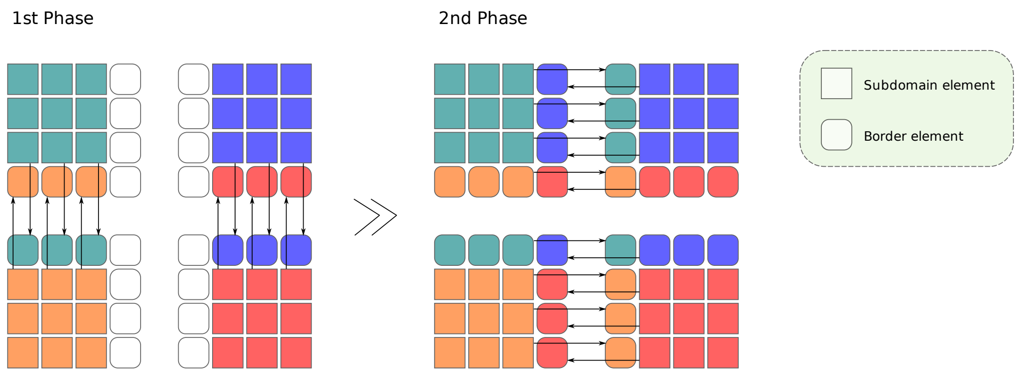


Figure 5.26 – Communication flows among blocks.

```
write_ghosts(rank, vert_border);
read_ghosts(vert_neighbor(rank));
write_ghosts(rank, horiz_border);
read_ghosts(horiz_neighbor(rank));
```

Figure 5.27 – Communication among threads managing a block partitioned domain.

Chapter 6

Benchmarks vs. Model Estimates

In this chapter we summarize the results obtained using the benchmark application described in chapter 5. The chapter starts describing the execution environment in Section 6.1. The test methodology is presented in Section 6.2, and results are shown and discussed in Section 6.3.

6.1 Experimental Environment

The experiment was run on a heterogeneous system composed by a multicore CPU and a multi-GPU system.
The quadcore microprocessor is based on the Nehalem microarchitecture, and the main features are listed in Table C.1. The multi-GPU system is the NVIDIA Tesla

Table 6.1 – CPU Features.

Architecture	Intel 64
Number of cores	4
Clock speed	3.20 GHz
SMT	Intel hyperthreading, which enables two threads per core.
Memory controller	Integrated DDR3 memory controller (three channels, 2 DIMMs/channel).
Connection to the I/O Hub	Intel QuickPath Interconnection (four 20-bit channels, 6.4 GT/s).
L1 cache	32 KiB L1 instruction and 32 KiB data cache per core.
L2 cache	256 KiB L2 cache per core.
L3 cache	8 MiB L3 Intel Smart Cache.

S1070 Computing System described in Section 2.3 (-400 configuration). The main characteristics are summarized in Table 6.2. The installed operating system is Ubuntu 9.04 with Linux kernel 2.6.28-11. The CPU application is compiled using gcc version 4.3.3 with option -O3. The GPU software is developed for CUDA version 2.1 and compiled with nvcc version 2.2[1].

6.2 Methodology

Every single execution of the PDE solver is aimed at solving a problem for $I = 100$ iterations. We organized the entire benchmark execution in three rounds. In the first round, we ran the solver on squared domains with dimension $N \times N$. In the second rounds, the solver was run on rectangular domains with dimensions $N/2 \times N$ and $N \times N/2$. Finally, in the last round, we ran the solver on squared domains varying the border size in the range [1-100].

Every single execution was run several times. The number of attempts per execution was mostly affected by the dimension of the domain (i.e. the larger the domain, the less the number of attempts). We noticed that, especially in between different executions, the first attempts produced ill-conditioned time results. For this reason, to estimate the central tendency of several tries of the same execution, we use the median, as the arithmetic mean is easily affected by the presence of outliers. Table 6.3 summarize the number of attempts per execution depending on the domain size and the benchmark round.

[1]CUDA 2.2 was released May 7, 2009. Since it does not introduce novelties relevant to our development, we decided to maintain the context of the previous 2.1 version.

Table 6.2 – NVIDIA Tesla S1070 Features.

Number of GPUs	4 Tesla T10, compute capability 1.3 (See Appendix A for a detailed list of features).
Number of SPs	960 (240 SP per GPU, 1.30 GHz)
Single precision peak performance	3.73 TFLOPS
Double precision peak performance	311 GFLOPS
Standard of precision	IEEE 754 for both single and double precision.
Device memory	16 GB (4 GB dedicated DRAM per GPU)
Memory interface	512-bit
GPU-device memory bandwidth	up to 408 GB/s.
Connection to the host	Through 2 PCIe channels pairwise shared. Two GPUs are connected to an NVIDIA switch. Every Switch is connected to a PCIe slot on the host through an NVIDIA Host Interconnection Card (HIC).
GPU-host bandwidth	up to 12.8 GB/s.
Run-time limit on kernels	No.

6.2.1 Collected Output Values

In the following, we indicate with G the number of GPUs, with I the number of iterations, with BS the border width, and with $time(a,i)$ the time required by the action a to take place at iteration i. In Section 3.2.2, we proposed a simple model for the communication between two GPUs. Now, to motivate our output acquisition, we want to extend the formulation to a multi-GPU system. Taking into account the description in Section 3.2, a multi-GPU system is the combination of a node and a set of GPUs. We propose the following formulation as a general model,

$$T_{multi-GPU} = T_{node}(m,p) + \max_{g \in G} \left(T^{g}_{kernel}(n,g) + T^{g}_{GPU-GPU}(n,p) \right) \ , \tag{6.1}$$

where T_{node} is the model of the specific node (e.g. a node of a multicomputer). The max operator is motivated by the fact that the several GPU contexts are executed in parallel.
Taking the model in (6.1) as a reference, the values collected from the execution output are:

Mean Kernel Time (MKT) Average time required to execute a kernel computation (both red and black kernels). Time in ms. Formally:

$$\max_{g \in G} \left(\frac{\sum_{i=0}^{I} (time(RSOR,i) + time(BSOR,i))}{I} \right)$$

Mean Synchronization Time (MST) Average time spent in synchronization during one iteration. Time in ms. Formally:

$$\max_{g \in G} \left(\frac{\sum_{i=0}^{I/\mathrm{BS}} time(synch,i)}{I/\mathrm{BS}} \right)$$

Table 6.3 – Number of attempts per execution depending on the domain size and the benchmark round.

Domain size	Round 1	Round 2	Round 3
$128 \leq N < 4000$	30	30	3
$4000 \leq N < 8000$	20	20	3
$8000 \leq N < 33000$	10	10	3
$33000 \leq N \leq 52000$	5	5	3

Mean Transfer Time (MTT) Average time spent transferring borders during one iteration. Time in ms. Formally:

$$\max_{g\in G}\left(\frac{\sum_{i=0}^{I/\mathrm{BS}} time(transfer, i)}{I/\mathrm{BS}}\right)$$

Mean Communication Time (MCT) Average time spent in communication during one iteration. Time in ms. Since MCT is an estimation of the time required by the whole communication code, MKT + MST $\leq$ MCT. Formally:

$$\max_{g\in G}\left(\frac{\sum_{i=0}^{I/\mathrm{BS}} time(communication, i)}{I/\mathrm{BS}}\right)$$

Elapsed Time (ET) Time in seconds required to solve the PDE for I iterations, with G GPUs, exchanging BS wide borders.

We want to remark that MKT, MST, MTT, and MCT refer to one iteration, that means that they are all measurements from the solver's core computation. The data transfers outside the core computation (i.e. the first and last transfers, responsible of moving the entire subdomains from and to the GPUs) as well as the time required to allocate space on the devices are not directly collected. This values are summarized in ET.

6.2.2 Timing

On the CPU, time was measured on the system-wide realtime clock using the *clock_gettime()* system call. The system call returns a value of type *struct timespec*, containing seconds and nanoseconds from the Epoch.

On the GPU, instead, time was retrieved using CUDA events. CUDA events are part of the CUDA API. They are recorded into CUDA call streams, and can be used to measure elapsed time for CUDA calls on the device with approximately 0.5 μs precision.

6.2.3 Helper Tools

Around the benchmark application, we developed a set of tools to help automating the creation of the set of domains, and the test phase. They are all described in Appendix D.

6.3 Results Discussion

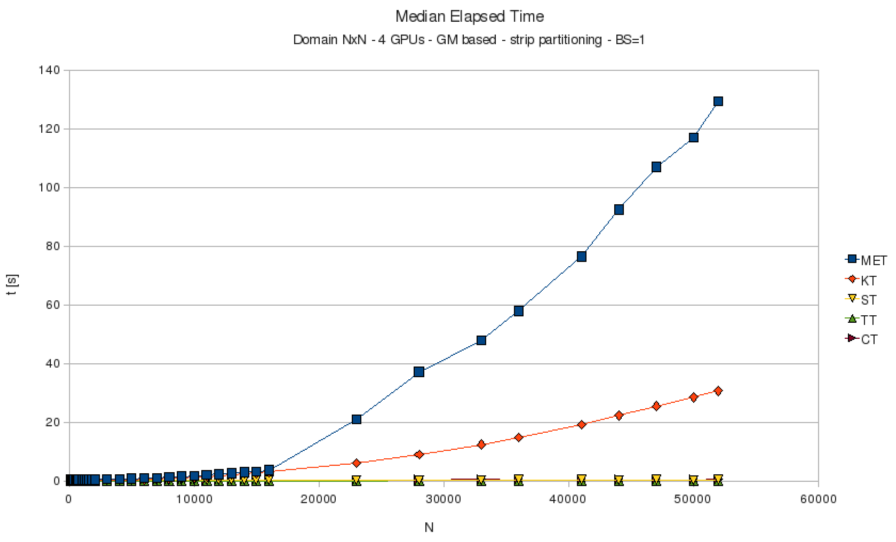
We describe our results from different perspectives, in order to discuss how the different variables exposed by our application can impact on performance factors. For the sake of clarity, in this section we report just those charts that we consider the most significant for our analysis. A wider collection of graphs can be found in Appendix E.

6.3.1 Domain Size and Data Partitioning

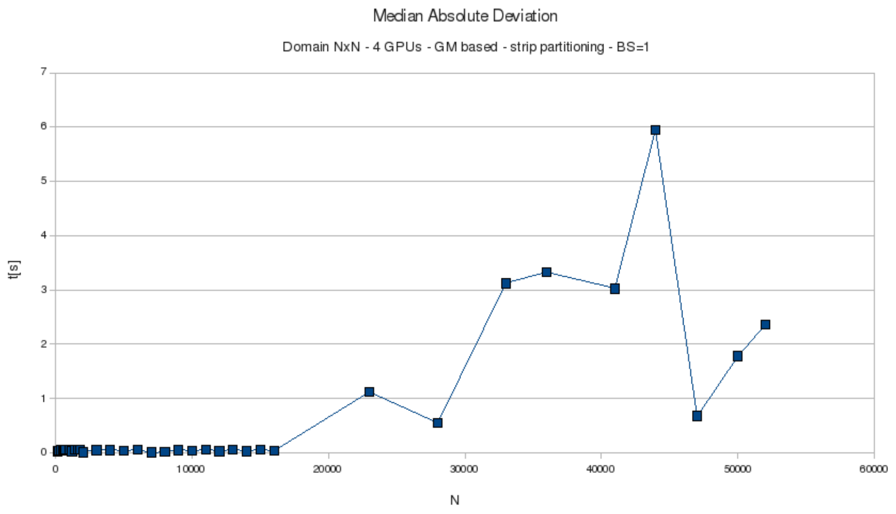
A first consideration comes from the evaluation of the execution time as the domain size increases. Aside the obvious time increase due to the elaboration of larger and larger domains, Figure 6.1 shows another relevant particular. In the graphs, the Kernel Time (KT), Synchronization Time (ST), Transfer Time (TT), and Communication Time (CT) are all obtained from the respective MKT, MST, MTT, and MCT measures. For this reason they refer, as we already mentioned, to the core computation only.
If we would exclusively consider the core computation, we may notice that the application is kernel bound. Indeed, while the size of the borders to transfer grows linearly, the area to process grows quadratically, becoming the most expensive overhead. However, the blue curve shows the median elapsed time, which takes into account the first and last transfers between host and GPUs, the device allocation time, and the first synchronization (that simply consists in signaling the condition variables related to the transfered memory areas after the first transfers). Allocation and synchronization represent a very small percentage of MET (Normally not over 10%). From this outer perspective then, the entire application becomes practically transfer bound, requiring, for large domains, up to three times the kernel time to transfer the entire subdomain areas.

In terms of data partitioning, the graphs in Figure 6.2 tell us that strip partitioning turns out as the best configuration, always outperforming the alternative blocked solution. Looking at the time outcomes shown in Figure 6.3, it seems that the reason is again related to the cost of transferring data outside the core computation. For large domain sizes, transferring blocks can cost almost four times more than transferring strips. In Section 5.3 we proposed a simple model of host-device communication. Such a model, however, appears rather inaccurate. We measured the time required by the *cudaMemcpy2D()* call to transfer different kinds of data block. Results are shown in Table 6.4. As we can notice, our model presents a large deviation from the real measurement. Transferring a block with dimension $n/2 \times k$ using the *cudaMemcpy2D()* takes much more (almost 50 times in the specific case of the table) than transferring a block of dimension $k \times n$. Such a result motivate the notable increase in time required when using block partitioning instead of strip partitioning.

However, we found a linear relation between the dimension of a block of contiguous elements expressed in gigabytes and the time required to transfer it. We acquired transfer times for different block dimensions and we calculate a a curve



(a) Median Elapsed Time (MET)

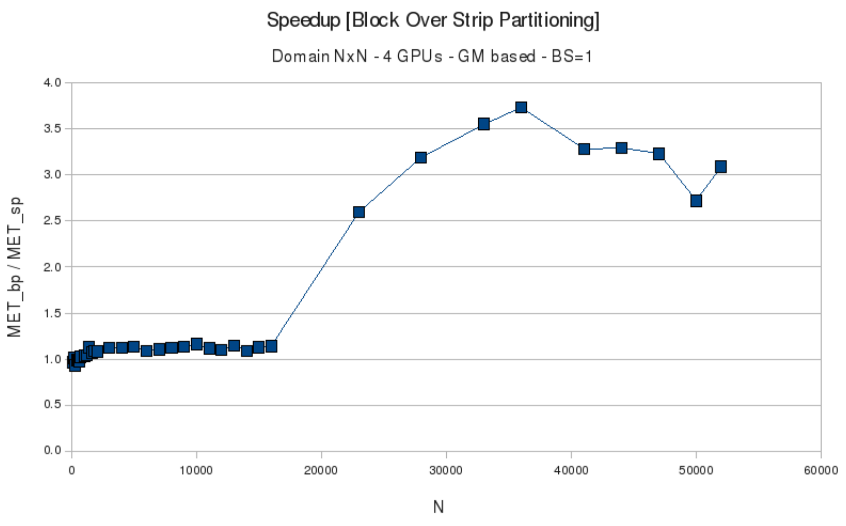


(b) Median Absolute Deviation

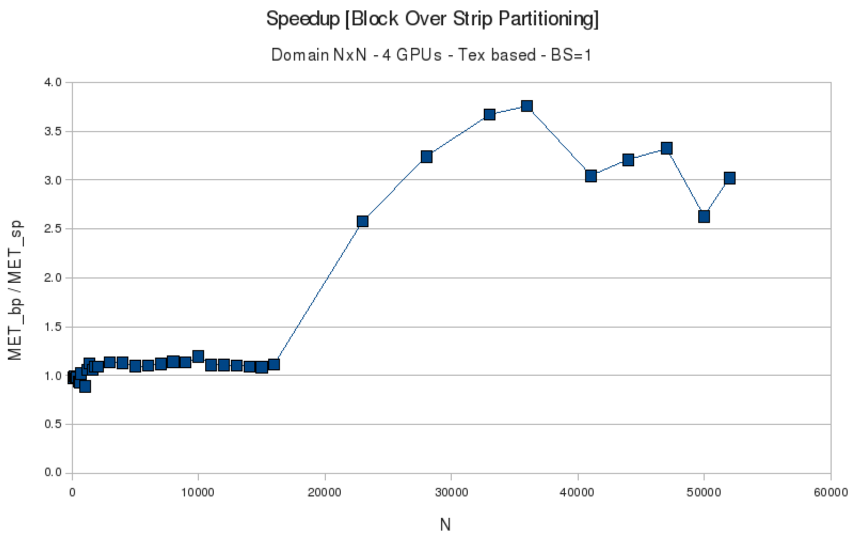
Figure 6.1 – Execution on domains NxN using: four GPUs, global memory based kernel, strip partitioning, and unitary border width.

Table 6.4 – Comparison between the linear model used in Section 5.3 and the real time required by the *cudaMemcpy2D()* function. The matrix involved has dimension 11585×11585. On the device, memory was allocated with *cudaMallocPitch()*.

K	N	Measured time [ms]	$K(t_{startup} + N * t_{word})$ [ms]
1	0	0.003	0.003
1	1	0.015	0.015
5	11585	0.145	3384.557
5	5792	0.049	1692.140
5792	5	7.443	1710.659

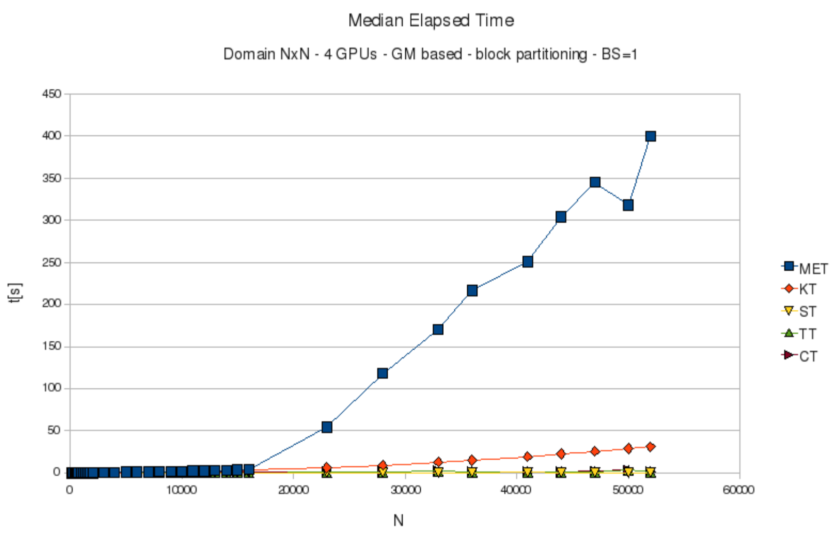


(a) Global memory based kernel.

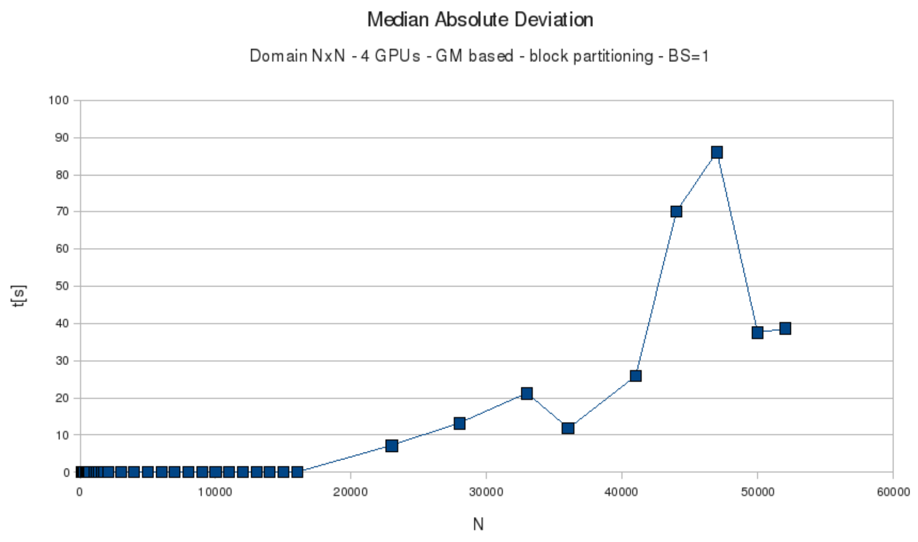


(b) Texture based kernel.

Figure 6.2 – Block over strip partitioning speedup. Execution on domains NxN using four GPUs and unitary border width.



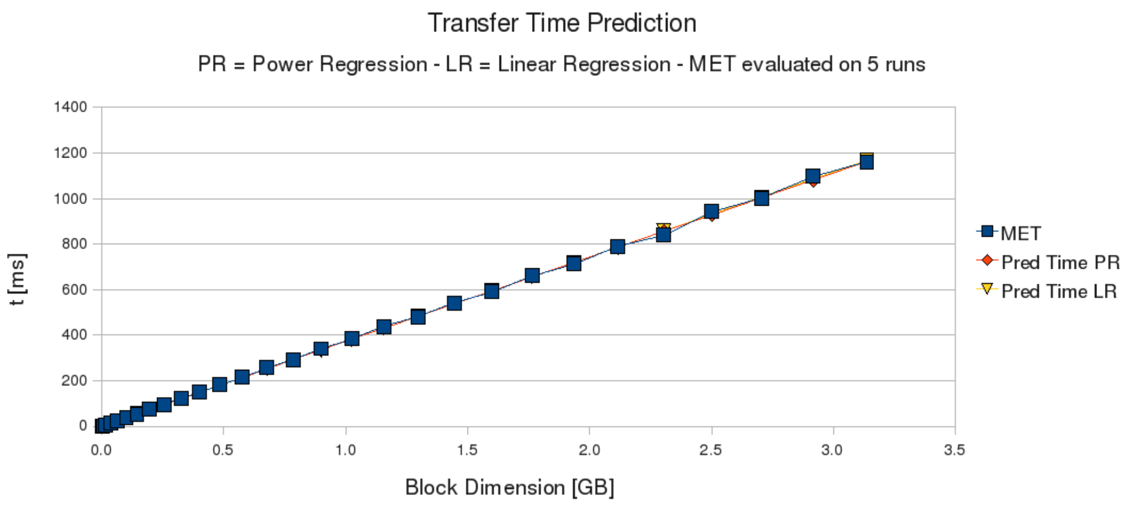
(a) Median Elapsed Time (MET)



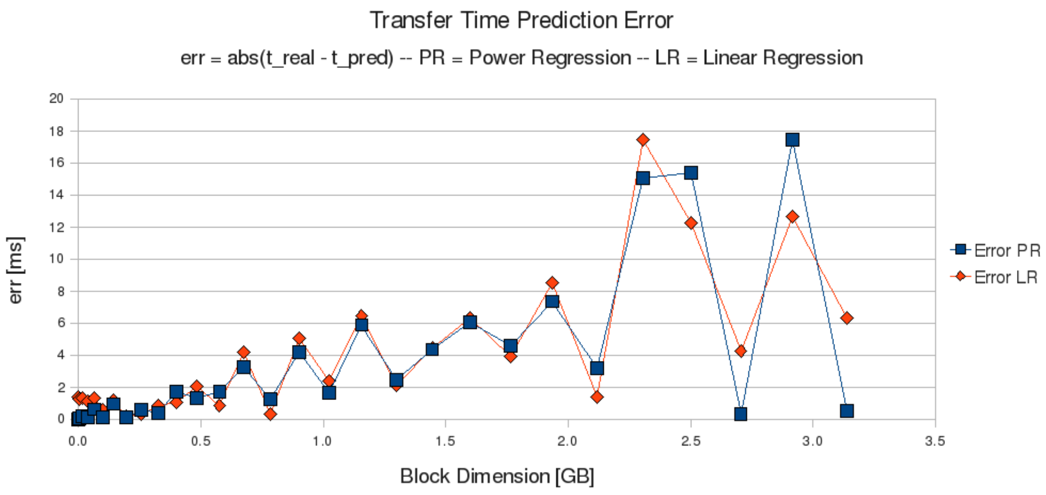
(b) Median Absolute Deviation

Figure 6.3 – Execution on domains NxN using: 4 GPUs, global memory based kernel, block partitioning, and unitary border width.

that could fit those values using the least squares method. The results are shown in Figure 6.4. The third curve represents a first attempt to find an approximation



(a) $T_{GPU-host}(n)$ prediction using statistical regressions.



(b) $T_{GPU-host}(n)$ prediction error.

Figure 6.4 – $T_{GPU-host}(n)$ prediction.

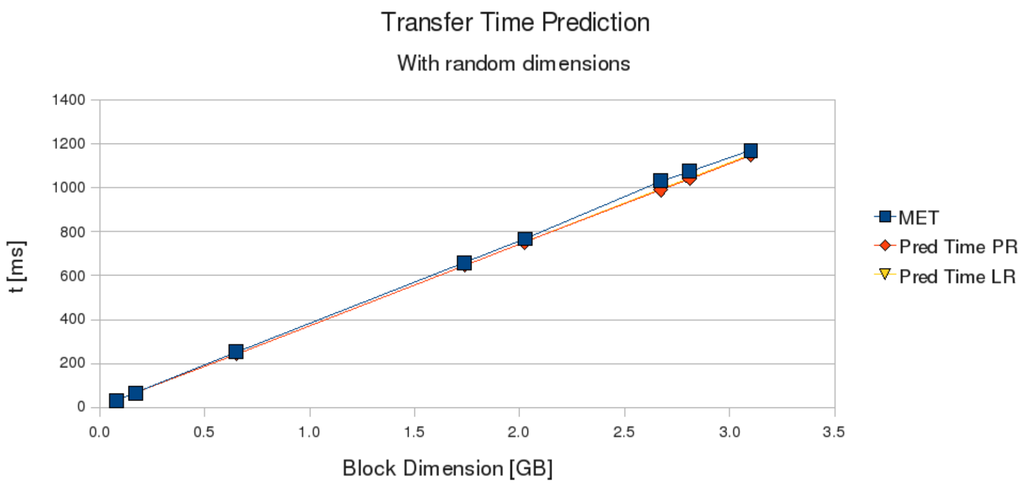
using a more general power regression method. Considering that we want to look for a general approximation of the transfer time, such as

$$T_{GPU-host}(n) = a \cdot n^b, \tag{6.2}$$

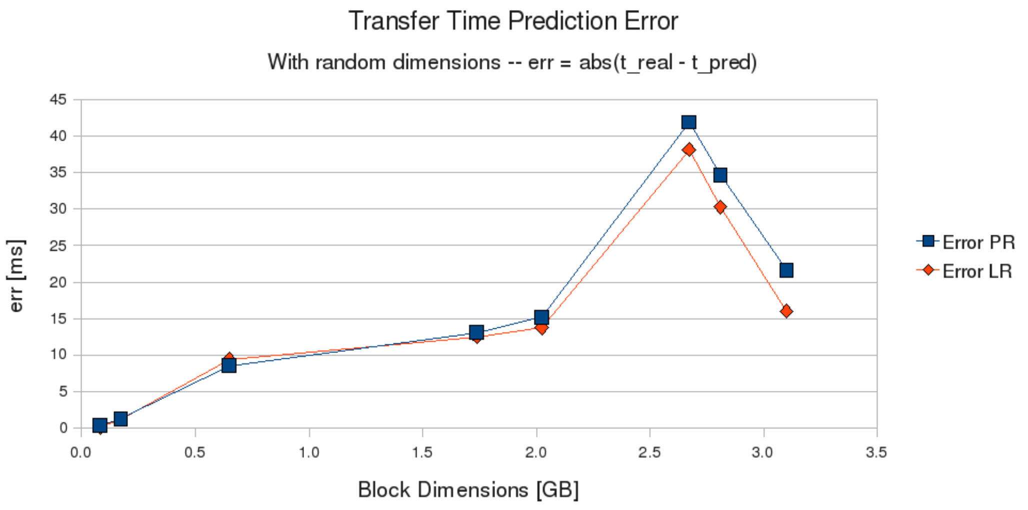
we could think to take a logarithmic scale of the function, obtaining the expression

$$\ln(T_{GPU-host}(n)) = \ln(a \cdot n^b) = \ln a + b \ln n. \tag{6.3}$$

At this point, it is possible to use linear regression in order to find $\ln a$ and b. Using this approach, we found a quasilinear exponent $b = 0.99$. This suggested to apply directly the least squares method to the initial set of data, obtaining the linear curve in Figure 6.5(a). The quasilinear curve obtained through power regression, is more precise for smaller domain sizes, while the linear curve appears more accurate for larger sizes. We used the curves with a smaller dataset of random dimensions to test their fitness, and the results are those shown in Figure 6.5. Using the same approach of linear regression, we similarly tested different Tesla-based GPUs installed on a different architecture, again obtaining acceptable approximations (see Appendix E.1.4). Such a result could be taken into account when there is a need to model the transfer of blocks of contiguous elements. However, it is important to



(a) $T_{GPU-host}(n)$ prediction fitting test.



(b) $T_{GPU-host}(n)$ prediction fitting test error.

Figure 6.5 – $T_{GPU-host}(n)$ prediction fitness test.

consider the nearness of the elements in the block as a strict requirement to make linear regression an acceptable predicting tool. As we can notice in Table 6.4, using different position patterns within the block may turn out in extremely different results. To define a better model for the memory copy function we would need a deeper knowledge of its implementation, but unfortunately, at the moment, such details are not available.

In any case, as our experience and many other recent findings prove [19], dynamic modeling techniques, based on statistical observations and methods, appears a better approach to define predicting models able to adapt to real systems' conditions than static models.

6.3.2 Domain Shape

Another variable we took into account is the domain shape. In our context we call Shape0 the rectangular shape of a domain with dimension $N/2 \times N$, and Shape2 the rectangular shape of a domain with dimension $N \times N/2$.
Figures 6.6 and 6.7 show the speedups for applications run with four GPUs. Speedups are calculated as the ratio of the median elapsed time using Shape0 to the median elapsed time using Shape2.

When using strip partitioned domains (Figure 6.6), there is not great benefit from using one shape or another. This because, as we noticed in the previous section analyzing the domain size, the core computation is kernel bound and, as recalled by Table 6.4, transferring contiguous bytes of memory is rather efficient. For block-partitioned domains, results are shown in Figure 6.7. In this case it is clearly visible how the high cost of horizontal transfers can degrade the application performance when using domain with Shape2.

6.3.3 Global Memory Vs. Texture Based Kernels

We focus now on the usage of different device memory areas. In Section 5.4.2, we discussed the possibility to load memory from device to shared memory using two possible approaches: through coalesced access to global memory, and fetching from texture memory. The first approach does not make use of intermediate cache, while the second one is supported by a 2D data cache.

We measured the time required by the application using both the mentioned approaches. Always based on the median elapsed time, Figure 6.8 shows the speedup as the ratio of the execution of global memory based kernels to the execution of texture based kernels. The shown results put the two choices practically at the same level, leading us to the conclusion that, in our context, using coalesced accesses is as efficient as exploiting space locality.

6.3.4 Number of GPUs

The effect of using more than one GPU is shown in Figure 6.9 just for the texture based case. The graphs underline that involving the most of the GPUs available is always an appropriate decision since from small dimensions of the domain ($N \times N > 3000 \times 3000 \approx 36$ MB). The curves show that using four GPUs can be up to 3.4X faster than using only one, and 1.8X faster than using two.

After $N = 16000$ however, the curves start exhibiting a drastic reduction in performance. We think that this effect can be produced by resource contentions. As

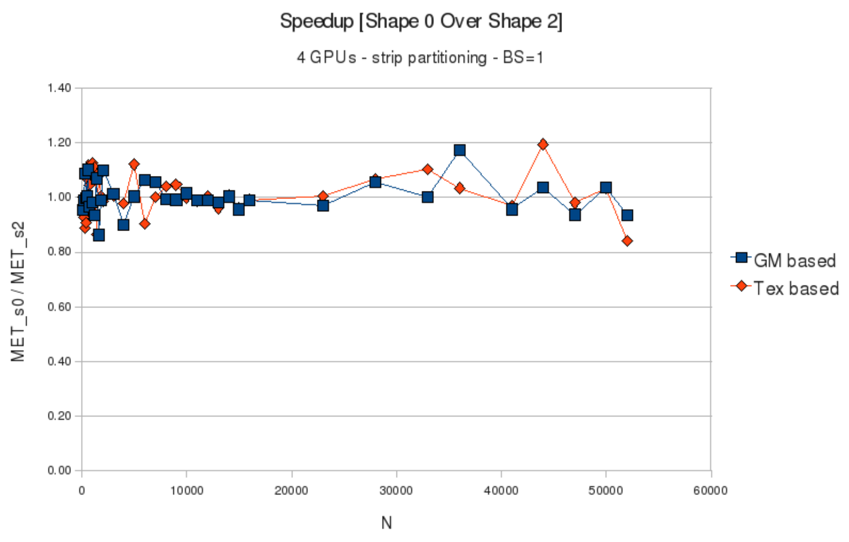


Figure 6.6 – Speedup of executions using Shape0 ($N/2 \times N$) domains over executions using Shape2 ($N \times N/2$) domains. Domains are strip-partitioned.

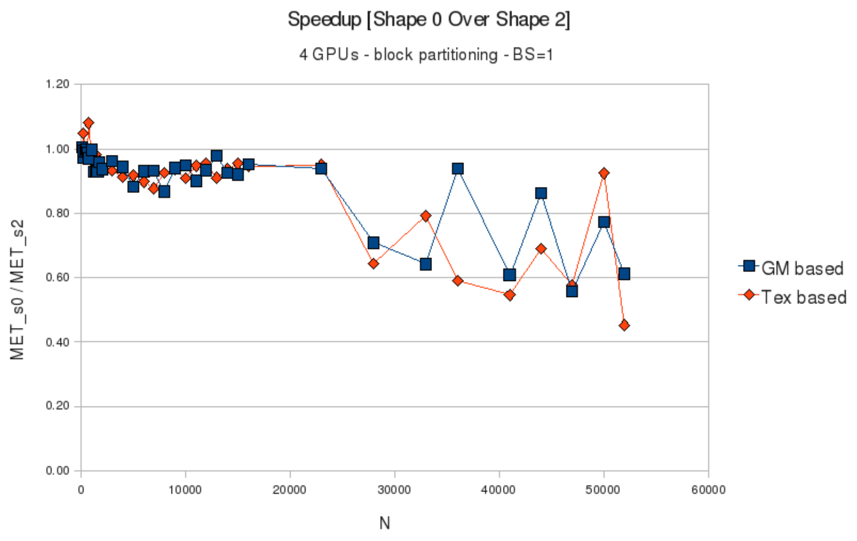


Figure 6.7 – Speedup of executions using Shape0 ($N/2 \times N$) domains over executions using Shape2 ($N \times N/2$) domains. Domains are block-partitioned.

described in Section 2.3, GPUs on the S1070 share pairwise the two PCIe channels. It is possible that, with a relevant traffic ($> 2GB$), the PCIe contention lowers the performance of two parallel transfers almost down to the performance of a single one. As a result, the execution time required by N GPUs gets closer to the time required by a subset of N.

6.3.5 Border Size

The approach described in Section 5.2 was shown an effective technique to overcome the bandwidth factor on SMP clusters when solving PDEs numerically. We ran the solver on some representative domains exchanging different borders with width in the range [1-100]. In [10] is reported that, for supercomputers with Infiniband interconnection, some minimal performance improvements were found using values of BS close to one. Since PCIe links are closer in bandwidth to Infiniband interconnections than not to Ethernet cables, we decided to have more test points in the neighborhood of $BS = 1$.

In Figures 6.11 we report results from the benchmark tests run using strip-partitioned, squared domains and texture-based kernels. Analyzing the graphs, we can deduce that the technique does not boost enough performance. The whole set of results can hardly approach a 10% of improvement (1.1X) with respect to the version based on unitary border width. Especially if we consider the tendency of the median absolute deviation to have a greater magnitude in proximity of $BS = 1$.

Thus, the empirical results bring us to a similar conclusion as in the supercom-

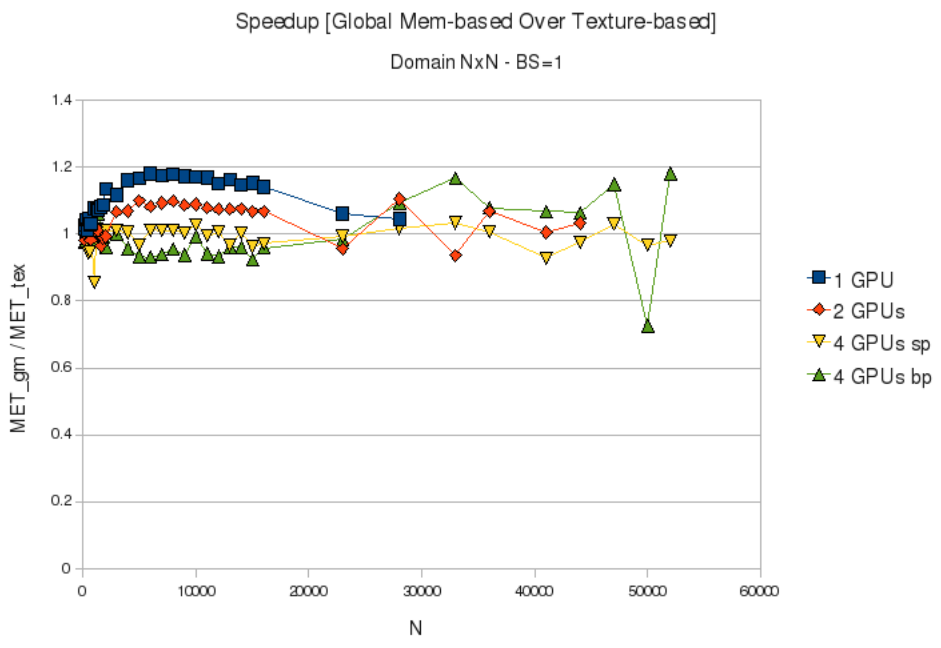
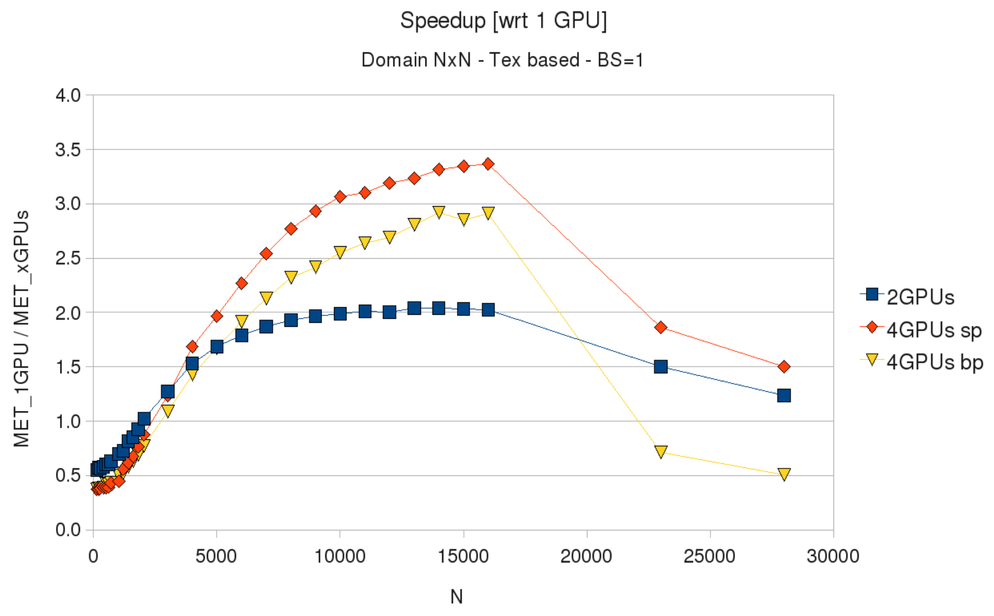


Figure 6.8 – Speedup of executions using global memory based kernels over executions using texture based kernels.

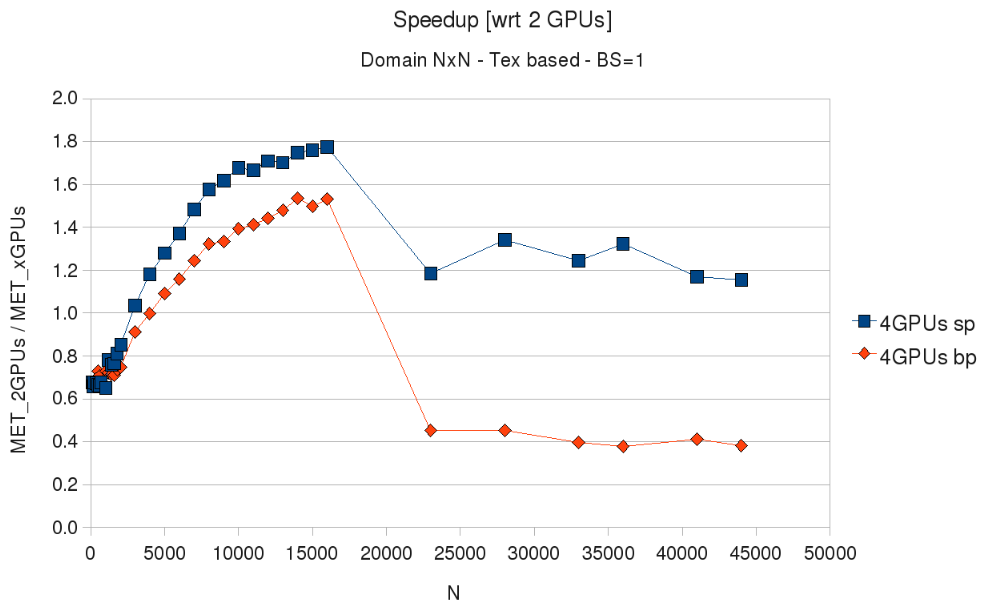
puters' case reported in [10], confirming our assumption that PCIe interconnections are fast enough to make the effect of the discussed method vanish.

6.3.6 Threads Synchronization

At design phase, we decided to base our communication on POSIX threads synchronization, in order to assess its impact on communication and compare it with the alternative option of message-passing libraries. The latter option was recalled in Section 3.2.2 for being responsible of around 70% of the communication overhead.



(a) Speedups with respect to one GPU.



(b) Speedups with respect to two GPUs.

Figure 6.9 – Speedups varying number of GPUs.

Figure 6.10 shows the impact of both synchronization and data transferring on communication per iteration. Contrarily to what expected, the impact of synchro-

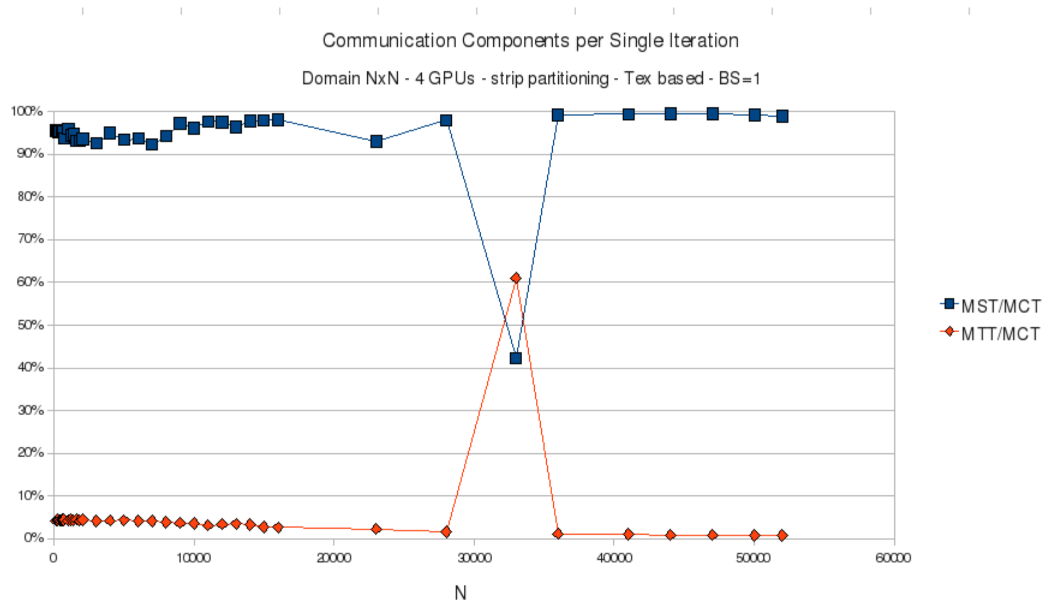
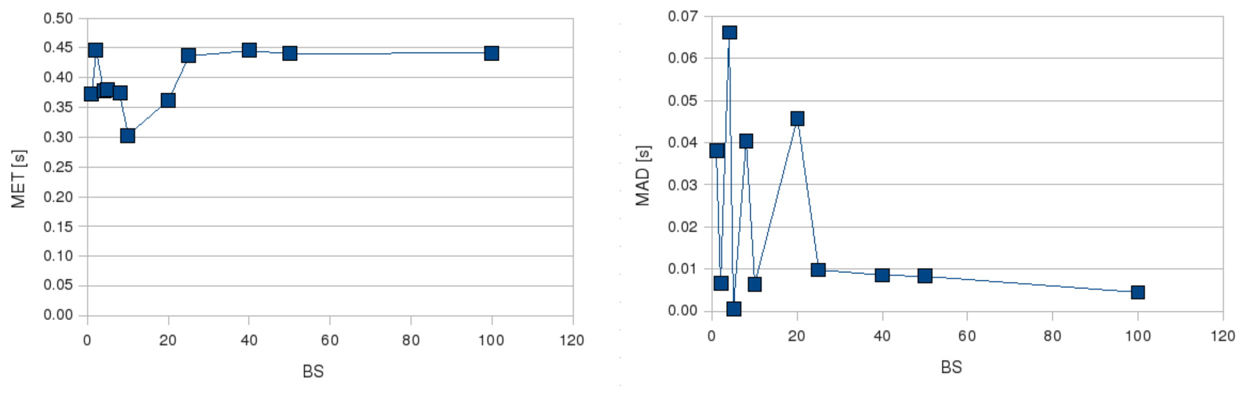
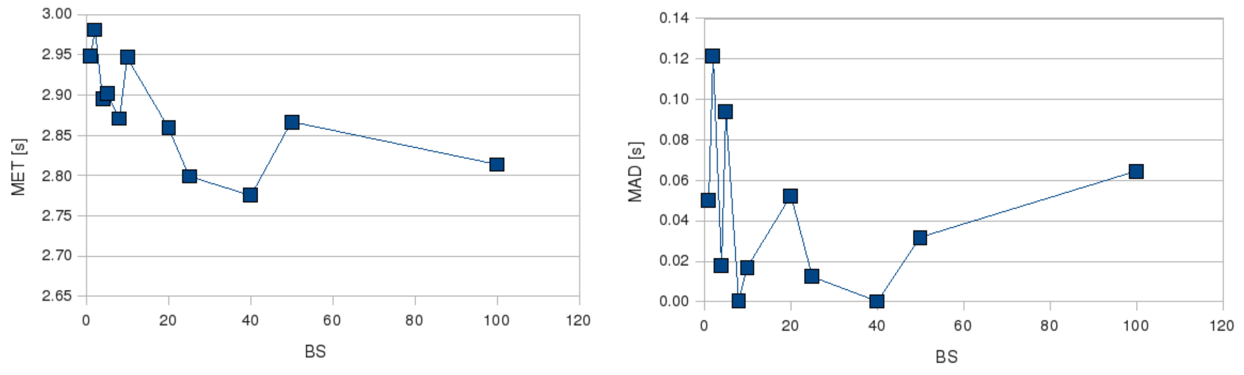


Figure 6.10 – Precentage of communication used for synchronization and data transfer.

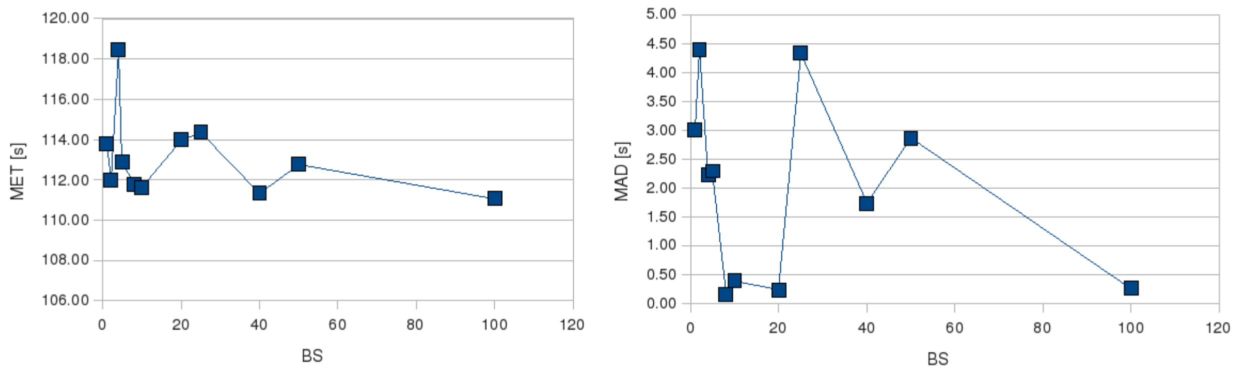
nization is quite relevant, practically dominating the overall communication. Even though this impact is almost negligible on large scale (the core computation is kernel bound), such an effect must be analyzed and controlled as it may become more relevant in the perspective of more powerful hardware and shorter distances between host and devices. Profiling, it seems that the delay is due to lock contentions on neighbor domain areas. We plan a deeper investigation in our future work.



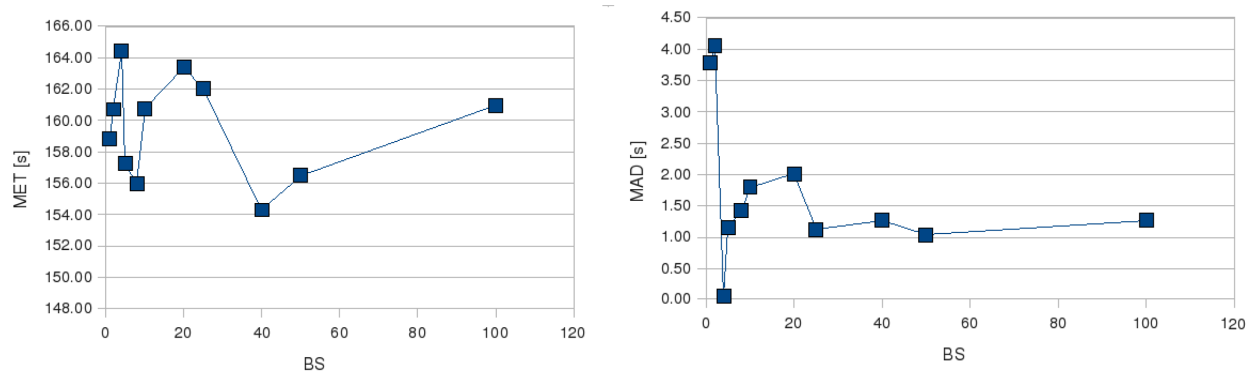
(a) Domain 1024x1024, strip partitioning, texture-based kernels, Median of 3 runs.



(b) Domain 10000x10000, strip partitioning, texture-based kernels, Median of 3 runs.



(c) Domain 44000x44000, strip partitioning, texture-based kernels, Median of 3 runs.



(d) Domain 52000x52000, strip partitioning, texture-based kernels, Median of 3 runs.

Figure 6.11 – Border size influence on performance using four GPUs.

Chapter 7

Conclusions and Future Work

With recent GPUs getting very close to the performance shown by the slowest Top-500 supercomputer only four years ago and the fastest one around ten years ago, GPU computing for high performance computing is generation a lot of interest. In this thesis, we investigated multi-GPU systems' performance factors, such as data volume dimensions, data partitioning techniques, number of GPUs, inter-GPU communication methods, and kernel design. In particular, we focused on NVIDIA's multi-GPU solutions, their current S1070, as system that has been and still is being deployed at HPC centers world wide, including Tokyo Technology University's Tsubame supercomputer which was ranked 29^th in the world when installed, and the new multi-GPU system recently ordered by GENCI in France. Our methodology and general results should, however, be applicable to most modern multi-GPU systems.

Modern multi-GPU systems could be likened to a distributed environment with a centralized interconnection fabric (the host side). Every computing node, i.e. a single GPU, cannot directly share memory with the other peers. Internally instead, a GPU is a highly parallel, multi-threaded environment able to schedule thousands of threads on hundreds of streaming cores. Our testbed was an NVIDIA S1070 with four GPUs connected to an Intel i7-extreme system with 12GB of memory. Considering the S1070 has 16 GB of memory, it would have been desirable for the host i7 system equal or more RAM, but this is rarely provided as motherboards servicing more than 12GB are much more expensive that those up to 12 GB.

The S1070 system was analyzed based on its specific hardware features and on possible analogies to fundamental parallel models, such as shared memory multiprocessors and multicomputers. The Poisson problem with Dirichlet boundary conditions was picked as our model problem since it does boarder exchanges of information common to a large class of application problems. Our results are hence most applicable to this class of problems. Based on all these assumptions, we defined a space of variables on which to analyze performance impacts. On top of this test space we developed a benchmark PDE solver tool. The main variables exposed by the tool were: domain size and shape, kind of data partitioning, number of GPUs, width of the borders to exchange, kernels to use, and kind of synchronization between the GPU contexts. Based on results obtained running the benchmark application varying such variables, we came to the following conclusions.

Application Bounds

Varying the domain's dimension up to 11 GB we found the application I/O bound. Transferring the domain from host to device memory took always the largest percentage of time, requiring up to three times the kernel time. Excluding the first and the last necessary data transfers, the core computation was instead kernel bound. Exchanging only the essential data during the computation, i.e. the subdomains' border, required no more than 10% of the total elapsed time.

Data Partitioning Impact

Since our test system only consisted of four GPUs, not surprisingly we found partitioning the data in vertical strips rather than blocked subdomains, to be the most effective. However, simple communication models, normally used in parallel computing theory, were not found suited for performance prediction on multi-GPU systems. Indeed, a general model would be very difficult to construct, since it would need to be based on an incredible number of specific architectural details related to different hardware, and hardware is always susceptible to changes.

On the other hand, statistical approaches were found to be more stable and adaptable to slight technological alterations. We found a linear relation between the size of contiguous data expressed in gigabytes and the time required to transfer such data between the host and the device. Through statistical regressions we found a model that was able to match our empirical curves. We tested the approach on different Tesla-based architectures with different PCIe interconnections, obtaining results that made, in our opinion, the involved statistical approach a good tool for communication modeling. Better specifications of the way communication is effectively organized at the API level would be necessary to develop more specialized models for GPU systems.

Domain Shape Impact

Different domain shapes had a very contained influence on performance ($< 1.2X$). Indeed, the amount of data to process is always the same, so it did not impact on the kernel time, while from a communication point of view, the borders to exchange are rarely large enough to make a real difference in transfer time.

Global Vs. Texture Memory-Supported Kernels

Our empirical results showed that, in our context, there was not an appreciable difference in performance when using cached versus non-cached memory, as long as the most important performance guidelines for designing access patterns to non-cached memory were followed.

Number of GPUs

Using the all the four GPUs available on the Tesla S1070, was always beneficial, performing up to 3.5X and 1.8X speedup with respect to one and two GPUs. However, performance decreased working with large data volumes possibly due to resource contention (e.g. with around 8 GB we got 1.2X with respect to two GPUs).

Reducing Communication Through Extra Computation

The technique proposed in [10] as beneficial for cluster-like systems did not produce any relevant improvement in our context. Varying the border width between 1 and 100 we never got better improvements than 1.1X. The interconnection system between host and devices did not exhibit a large latency, making our case much closer to the one documented for supercomputers.

GPU Contexts Synchronization

Synchronization between GPU contexts through Pthread condition variables took a relatively large percentage of the communication during the core computation. This is similar to what was previously documented for shared-memory based, message-passing libraries [17]. Probably due to lock contentions, the measured delay deserves special attention in a more general context, as described in the future work.

7.1 Future Work

Benchmarking is not an easy task. To reach useful conclusions about a factor is often necessary to draw a wide angle of it from different points of view. Unfortunately, not everything we thought to do could properly fit in our limited time frame. For this reason, we want to illustrate in the present section what we consider possible development directions for our work.

7.1.1 An Improved Standard Framework for Benchmarking

On December 8, 2008, the consortium Khronos Group[1] approved for public release the first version specification of the Open Computing Languange (OpenCL) [2]. OpenCL is the first attempt of a standard language for heterogeneous systems sustained by a wide list of companies and academics. As more and more vendors will provide support for OpenCL, providing unified methods and models for comparing the performance across a multitude of heterogeneous architectural proposals is likely to become a crucial goal of the GPGPU community.

The tool we developed for our tests can be considered a premature stage of what could become a framework devoted to platform-independent, multi-GPU benchmarking. Also considering what is missing to meet the initial requirements, we would like to point out at least four directions:

[1] http://www.khronos.org/

- More attention must be paid to decouple the kernel logic from the synchronization logic, so to allow an easier and more independent design and analysis of both.
- More focus on precision. As described in [27], the use of different precision standards, can also have a certain impact on performance. In a PDE solver context, introducing exit conditions based on proper approximations of the sought solutions can be a possible way to investigate eventual delays introduced by graphics hardware's precision.
- 3D modeling. Our framework could be extended to the third dimension, which is, in a way, naturally supported by graphics hardware. In a 3D context, there are some aspects that do not figure in the 2D case. For example, the requirement of exchanging not only borders but also surfaces introduces asymmetries in communication that would be important to examine.
- Platform independent. In order to make the framework aim at different platforms, a porting to OpenCL must be planned.

7.1.2 A Fast Communication Framework for GPUs

During our development we would have found beneficial the presence of an efficient synchronization/interconnection library between the GPUs. For highly parallel systems, such as supercomputers, the presence of efficient MPI communication routines is vital to develop efficient software. Likewise, the presence of a library for fast communication in a multi-GPU or GPU cluster context may improve applications' performance providing them with efficient implementations of communication primitives. To better support communication modeling, we think that the library should be based on some existing standard specification.

7.1.3 Large Scale Perspective

A next challenging goal should be the analysis of performance factors at a higher level of complexity. GPU clusters are heterogeneous systems composed by several multi-GPU nodes. In such systems, different GPUs are interconnected through a two-level communication network.

7.1.4 Green Computing

The interest of the international community for environmental friendly systems is a clear sign that speed is no longer the only requirement in supercomputing. With the introduction of the Green500 list[2], vendors are challenged to optimize the ratio performance/Watt instead of the only speed factor. This call for energy efficiency must be especially taken into account when dealing with GPU-based systems, since GPUs may be the largest power consumer components. Benchmarking energy consumption is an interesting endeavor. It would require to to adopt specialized hardware to monitor power usage, as there is no software support for such an activity.

[2]http://www.green500.org

Bibliography

[1] "Message Passing Interface Forum," http://www.mpi-forum.org/.

[2] "OpenCL," http://www.khronos.org/opencl/.

[3] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Corporation, Tech. Rep., Dec. 2008.

[4] D. Blythe, "Rise of the graphics processor," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 761–778, May 2008.

[5] W. Cheney and D. Kincaid, *Numerical Mathematics and Computing.* Brooks/Cole Publishing Company, 1999.

[6] R. Eidissen, "Utilizing GPUs for Real-time Visualization of Snow," Master's thesis, Norwegian Univ. of Science and Technology, Feb. 2009.

[7] A. C. Elster, "Parallelization issues and particle-in-cell codes," Ph.D. dissertation, Cornell University, Aug. 1994.

[8] R. Gerber, A. J. C. Bik, K. B. Smith, and X. Tian, *The Software Optimization Cookbook: High-performance Recipes for IA-32 Platforms.* Intel Press, 2006.

[9] M. T. Heath, *Scientific Computing - An Introductory Survey.* McGraw-Hill, 2002.

[10] R. Holtet, "Communication-reducing Stencil-based Algorithms and Methods," Master's thesis, Norwegian Univ. of Science and Technology, Jul. 2003.

[11] A. Iserles, *A First Course in the Numerical Analysis of Differential Equations.* Cambridge University Press, 1996.

[12] R. E. Jensen, "Techniques and Tools for Optimizing Code on Modern Architectures - A Low Level Approach," Master's thesis, Norwegian Univ. of Science and Technology, May 2009.

[13] E. Kreyszig, *Advanced Engineering Mathematics.* John Wiley & Sons, 2006.

[14] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, Sep. 1979.

[15] L. C. Larsen, "Utilizing GPUs on cluster computers," Norwegian Univ. of Science and Technology, Tech. Rep., 2006.

[16] W. R. Mark, R. S. Glanville, K. Akeley, and M. Kilgard, "Cg: a system for programming graphics hardware in a C-like language," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 896–907, 2003.

[17] P. Micikevicius, "3d finite difference computation on GPUs using CUDA," in *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units*, vol. 383, March 2009, pp. 79–84.

[18] T. Natvig, "Automatic Optimization of MPI Applications - Turning Synchronous Calls Into Asynchronous," Master's thesis, Norwegian Univ. of Science and Technology, Jan. 2006.

[19] T. Natvig and A. C. Elster, "Using context-sensitive transmission statistics to predict communication time," in *PARA 2008, LNCS 2009*, A. C. E. *et al.*, Ed. Springer, to be published.

[20] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, March/April 2008.

[21] *CUDA - CUBLAS Library 2.1*, http://www.nvidia.com/object/cuda_develop.html, NVIDIA Corporation.

[22] *CUDA - CUFFT Library 2.1*, http://www.nvidia.com/object/cuda_develop.html, NVIDIA Corporation.

[23] *NVIDIA CUDA 2.1 Programming Guide*, http://www.nvidia.com/object/cuda_develop.html, NVIDIA Corporation.

[24] *NVIDIA CUDA 2.1 Reference Manual*, http://www.nvidia.com/object/cuda_develop.html, NVIDIA Corporation.

[25] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[26] P. S. Pacheco, *Parallel Programming with MPI.* Morgan Kaufmann, 1997.

[27] D. G. Spampinato and A. C. Elster, "Linear optimization on modern GPUs," in *Workshop on Multi-Threaded Architectures and Applications as part of the 23rd IEEE International Parallel & Distributed Processing Symposium*, Rome, Italy, May 2009, (CDROM) ISSN: 1530-2075, ISBN: 978-1-4244-3750-4.

[28] A. S. Tanenbaum, *Modern Operating Systems, 3rd edition.* Prentice Hall, 2008.

[29] B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice Hall, 2005.

[30] S. Yang and M. K. Gobbert, "The optimal relaxation parameter for the SOR method applied to the Poisson equation in any space dimensions," *Applied Mathematics Letters*, vol. 22, pp. 325–331, 2009.

[31] D. M. Young, *Iterative Solution of Large Linear Systems.* Dover, 2003.

Glossary

API	Application Program Interface. A set of calling conventions defining how a service is invoked through a software package.
BLAS	Basic Linear Algebra Subprograms. A standard set of public domain mathematical subroutines that perform linear algebra operations.
CUDA	Compute Unified Device Architecture. it is a parallel computing environment developed by NVIDIA for their GPUs.
CPU	Central Processing Unit.
FLOPS	FLoating point Operations Per Second.
GPU	Graphics Processing Unit.
GPGPU	General-purpose computing on graphics processing units.
HCI	Host Interconnection Card.
HPC	High Performance Computing.
IEEE	Institute of Electrical and Electronics Engineers. A non-profit organization, IEEE is the world's leading professional association for the advancement of technology.
MIU	Multithreaded Instruction Unit. It is a Streaming Multiprocessor's unit dedicated to create, schedule, and manage CUDA threads.
MPI	Message Passing Interface. It is the de facto standard message-passing library specification.
NPTL	Native POSIX Thread Library. It is an efficient implementation of the POSIX Threads standard.
NUMA	Non-Uniform Memory Access. It is a shared memory architecture used in parallel computers. NUMA means that it will take longer to access some regions of memory than others.
OpenCL	Open Computing Language. It is a framework for executing programs across heterogeneous platforms.
PDE	Partial Differential Equation.
PCIe	Peripheral Component Interconnect Express. PCI Express architecture is an industry standard high-performance, general-purpose serial I/O interconnect designed for use in enterprise, desktop, mobile, communications and embedded platforms.

POSIX	Portable Operating System Interface for Unix. is the name of a family of related standards specified by the IEEE to define the API, along with shell and utilities interfaces for software compatible with variants of the Unix operating system, although the standard can apply to any operating system.
SIMD	Single Instruction Multiple Data. Part of the Flynn's taxonomy of computer architectures. In this form, a large group of simple processors all perform the same task in parallel, each with different data.
SIMT	Single Instruction Multiple Threads. Meant to refer to SIMD, it is the computing paradigm of NVIDIA Tesla-based GPU.
SLI	Scalable Link Interface. NVIDIA technology used in graphics mode to allow two or more GPUs to work together to produce a single graphical output from different input images processed in parallel.
SM	Streaming Multiprocessor. It is a computing unit within an NVIDIA Tesla-based GPU. It is composed by eight Streaming Processors.
SMP	Symmetric Multiprocessor. It is a shared memory multiprocessor with a numerical symmetry between the number of processors and the number of memory modules.
SOR	Successive Overrelaxation. It is an efficient method of solving a linear system of equations.
SP	Streaming Processor. It is a computing unit within a Streaming Multiprocessor. It executes CUDA threads.
UMA	Uniform Memory Access. It is a shared memory architecture used in parallel computers. All the processors in the UMA model share the physical memory uniformly.

Appendix A

Specifications for Compute Capability 1.3

The following is a list of features associated to devices with compute capability 1.3:

- The maximum number of threads per block is 512;
- The maximum size of the x-, y-, z-dimension of a thread block are 512, 512, and 64, respectively;
- The maximum size of each dimension of a grid of thread blocks is 65535;
- The warp size is 32 threads;
- The number of registers per SM is 16384;
- The amount of shared memory available per SM is 16 KB organized into 16 banks;
- The total amount of constant memory is 64 KB;
- The cache size for constant memory is 8 KB per SM;
- The cache size for texture memory varies between 6 and 8 KB per SM;
- The maximum number of active blocks per SM is 8;
- The maximum number of active warps per SM is 32;
- The maximum number of active threads per SM is 1024;
- The limit on kernel size is 2 million PTX assembly code instructions;
- For a texture reference bound to linear memory, the maximum width is 2^{27};
- Support for atomic functions in shared and global memory;
- Support for warp vote functions;
- Support for double-precision floating-point numbers;
- Each SM is composed of eight SP, so that a SM is able to process one warp in four clock cycles.

Appendix B

Code Samples

To carry out our analysis, we implemented a benchmark application to solve Dirichlet problems using a 5-point stencil, red-black SOR, PDE solver. Here we report relevant piece of code from the benchmark implementation. The PDE solver's codename is Tetra.

B.1 Benchmark Entry Point

```
/**
 * Tetra is a parallel PDE Solver for Dirichlet Problems.
 * The PDE is solved using a multi-GPU supported, 5-points stencil,
 * Red-Black SOR.
 **/

#include <cstdlib>
#include <sstream>
#include <getopt.h>

#include <tetra.h>

//command-line options (short version)
const char * const short_options = "ht:m:n:f:i:b:k:o:s";

//command-line options (long version)
const struct option long_options[] = {
  {"help",         0,  NULL, 'h'},
  {"threads",      1,  NULL, 't'},
  {"rows",         1,  NULL, 'm'},
  {"columns",      1,  NULL, 'n'},
  {"file",         1,  NULL, 'f'},
  {"iterations",     1,  NULL, 'i'},
  {"border_size", 1,  NULL, 'b'},
  {"kernel",       1,  NULL, 'k'},
  {"omega",      1,  NULL, 'o'},
  {"save_image",     0,  NULL, 's'}
};

/*
 * MAIN FUNCTION
 */

int main(int argc, char * argv[])
{
  int threads, m, n, iterations, bs, kernel, save_img = 0;
  float omega;
  string filename;

  int next_opt, opt_counter = 0;
  /* Code to extract command-line options omitted */

  if(bs > iterations)
  {
    cerr << "Border_size can't be greater than iterations. Abort."
      << endl;
    exit(2);
  }

  //Instianciate the solver
  Tetra t(filename, m, n);

  //Solve the problem
  t.solve(kernel, &omega, bs, iterations, threads);

  //Eventually save output image
  if(save_img)
  {
    t.saveImage("...title...");
  }

  exit(EXIT_SUCCESS);
}
```

B.2 Tetra PDE Solver

B.2.1 Tetra Class

```
  class Tetra {
    //Pointer to the domain
    float *data;
    //dim_x = columns, dim_y = rows
    int dim_x, dim_y;
    //Domain file pointer
    int fd;
    public:
      Tetra(string ifilename, int m, int n);
      ~Tetra();
      int solve(int solver, float *omega, int border_size,
        int iterations, int threads);
      int saveImage(string title);
      void printArea(int x_start, int x_end, int y_start, int y_end);
      void printGrid();
  };
```

B.2.2 PThread-Base Synchronization Header.

Samples from *psync.cu*

```
#include <pthread.h>
#include <math.h>
#include <sys/types.h>
#include <time.h>

#include <pde_kernels.cu>

struct Params
{
  char solver;
  float omega;
  char threads;
  int device;
  float *grid;
  int n;
  int h;
  int stride;
  int border_size;
  int iterations;
};

pthread_mutex_t *mutex;

pthread_cond_t *rCond, *wCond;
int *rAcc, *wAcc;

//////////////// CUDA WRAPPER ////////////////////
extern "C"
  int call_solver(
    int solver, float *omega, float *grid, int m, int n,
    int iterations, int threads, int border_size
  );
//////////////////////////////////////////////////

/*
 * getTime
 * Based on the timespec format. Returns end-start in seconds.
 *
 */

double getTime(timespec start, timespec end)
{
  long nsec = end.tv_nsec-start.tv_nsec;
  if(nsec < 0)
  {
    nsec = 1E9 + nsec;
    end.tv_sec -= 1;
  }
  return (double)(end.tv_sec - start.tv_sec) + (double)(nsec * 1E-9);
}

/*
 * selectTextureToBind
 * Select the right texture reference depending on the GPU context.
 *
 */

size_t selectTextureToBind(int device, int step, float *devPtr, size_t size)
{
  size_t offset;
  switch(device)
  {
    case 0:
      cudaBindTexture(&offset, texRef_0, devPtr, size);
      break;
    case 1:
      cudaBindTexture(&offset, texRef_1, devPtr, size);
      break;
    case 2:
      cudaBindTexture(&offset, texRef_2, devPtr, size);
      break;
    case 3:
      cudaBindTexture(&offset, texRef_3, devPtr, size);
      break;
    default:
      printf("Eventually you must add textures to bind to...\n");
  }

  return offset;
}
```

B.2.3 Strip Partitioning - Computation and Communication

Samples from *psync_strips.cu*

```
/*
 * tetra_thread
 *  params - contains the parameters required to setup a partial solver (i.e. the
     solver running on one of the GPU and solving
 *  computing a specific portion of domain).
 */

void * tetra_thread(void *params)
{
  //Cast the cookie pointer to the right type
  struct Params *p = (struct Params *)params;

  struct timespec h_start, h_end;

  clock_gettime(CLOCK_REALTIME, &h_start);

  selectDevice(p->device);

  clock_gettime(CLOCK_REALTIME, &h_end);

  dev_sel_time[p->device] = getTime(h_start, h_end);

  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);

  float r_kernel_time = 0, b_kernel_time = 0, temp_time;

  //Setup subdomain processing
  float *dgrid;
  int offset, height;
  size_t pitch;

  //Total number of rows counting borders. GPUs in the middle
  //deal with two sides.
  if((p->device == 0) || (p->device == p->threads - 1))
    height = p->h + (p->border_size);
  else
    height = p->h + (2 * p->border_size);

  //Offset to the beginning of the subdomain
  if(p->device == 0)
    offset = 0;
  else
    offset = p->border_size * p->n;

  //Allocate memory on the device
  cudaMallocPitch((void **)&dgrid, &pitch, p->n * sizeof(float), height);

  //Kernel Execution Configuration
  int grid_x, grid_y, local_w;

  //Number of threads per threadblock
  local_w = 112;

  //Number of threadblocks per grid. Strip partitioning, the
  //horizontal dimension doesn't change.
  //Blocks shift of (local_w-2) elements horizontally.
  grid_x = p->n/(local_w - 2);
  if(grid_x * (local_w - 2) < p->n) grid_x++;

  dim3 blockDim(local_w);

  //First transfer. Entire subdomain+borders.
  cudaEventRecord(start, 0);

  cudaMemcpy2D(dgrid, pitch, p->grid - offset, p->n * sizeof(float),
    p->n * sizeof(float), height, cudaMemcpyHostToDevice);

  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&temp_time, start, stop);
  ext_cpy_time[p->device] = temp_time;
  cpy_time[p->device] = 0;
  comm_time[p->device] = 0;

  clock_gettime(CLOCK_REALTIME, &h_start);

  //Increase read counters.
  if(p->device == 0)
  {
    pthread_mutex_lock(&mutex[1]);
      rAcc[1]++;
    pthread_mutex_unlock(&mutex[1]);
    pthread_cond_signal(&rCond[1]);
  }
  else if(p->device == p->threads - 1)
  {
    pthread_mutex_lock(&mutex[2 * p->device - 2]);
      rAcc[2 * p->device - 2]++;
    pthread_mutex_unlock(&mutex[2 * p->device - 2]);
    pthread_cond_signal(&rCond[2 * p->device - 2]);
  }
  else
  {
    pthread_mutex_lock(&mutex[2 * p->device - 2]);
      rAcc[2 * p->device - 2]++;
    pthread_mutex_unlock(&mutex[2 * p->device - 2]);
    pthread_cond_signal(&rCond[2 * p->device - 2]);

    pthread_mutex_lock(&mutex[2 * p->device + 1]);
      rAcc[2 * p->device + 1]++;
    pthread_mutex_unlock(&mutex[2 * p->device + 1]);
    pthread_cond_signal(&rCond[2 * p->device + 1]);
  }
  clock_gettime(CLOCK_REALTIME, &h_end);
  ext_sync_time[p->device] = getTime(h_start, h_end);
  sync_time[p->device] = 0;

// CORE COMPUTATION - Begin //////////////////////////

  //So to be able to iterate with border_size == 0
  int b = p->border_size > 0 ? p->border_size : 1;

  for(int i = 0; i < p->iterations/b; i++)
  {
    int border = b;
    //k is an offset used to point to the beginning of the area to process
    int k = 0;
    while(border > 0)
    {
      //Compute the vertical dimension of the grid.
      //Blocks shift with window size 2.
      grid_y = (2 * height - 4)/sizeof(float) + (height) % 2;
      dim3 gridDim(grid_x, grid_y);

      switch(p->solver)
      {
        case 0:
        {
          /* apply R/B SOR, using global mem for fetching data. */
          cudaEventRecord(start, 0);

          //Red kernel
          rsor_gm <<<gridDim, blockDim, 4 * local_w * sizeof(float)>>> (
            dgrid + k * (pitch/sizeof(float)), p->n, height, p->omega,
            pitch/sizeof(float), (p->h * p->device) - (p->device > 0) * border,
            p->device);

          cudaEventRecord(stop, 0);
          cudaEventSynchronize(stop);
          cudaEventElapsedTime(&temp_time, start, stop);
          r_kernel_time += temp_time;

          cudaEventRecord(start, 0);

          //Black kernel
          bsor_gm <<<gridDim, blockDim, 4 * local_w * sizeof(float)>>> (
            dgrid + k * (pitch/sizeof(float)), p->n, height, p->omega,
            pitch/sizeof(float), (p->h * p->device)-(p->device > 0) * border,
            p->device);

          cudaEventRecord(stop, 0);
          cudaEventSynchronize(stop);
          cudaEventElapsedTime(&temp_time, start, stop);
          b_kernel_time += temp_time;

          break;
        }
        case 1:
        {
          /* apply R/B SOR, using a texture to fetch data */
          size_t tex_offset = selectTextureToBind(p->device, i, dgrid + k * (pitch/
              sizeof(float)),
            height * pitch);

          cudaEventRecord(start, 0);

          //Red kernel
          rsor_lintex <<<gridDim, blockDim, 4 * local_w * sizeof(float)>>> (
            dgrid + k * (pitch/sizeof(float)), p->n, height, p->omega,
            pitch/sizeof(float), (p->h * p->device) - (p->device > 0) * border,
            tex_offset/sizeof(float), p->device);

          cudaEventRecord(stop, 0);
          cudaEventSynchronize(stop);
          cudaEventElapsedTime(&temp_time, start, stop);
          r_kernel_time += temp_time;

          cudaEventRecord(start, 0);

          //Black kernel
          bsor_lintex <<<gridDim, blockDim, 4 * local_w * sizeof(float)>>>(
            dgrid + k * (pitch/sizeof(float)), p->n, height, p->omega,
            pitch/sizeof(float), (p->h * p->device) - (p->device > 0) * border,
            tex_offset/sizeof(float), p->device);

          cudaEventRecord(stop, 0);
          cudaEventSynchronize(stop);
          cudaEventElapsedTime(&temp_time, start, stop);
          b_kernel_time += temp_time;

          break;
        }
      }

      border--;
      //increase k everywhere but on device 0
      k += (p->device > 0);
      //Decrease on outer devices and subtract 2 inside
      height -= 1 + ((p->device > 0) && (p->device < p->threads - 1));
    } //End border consuming while

    //Restore height
    height += b * (1 + ((p->device) && (p->device < p->threads - 1)));
```

```
//***** Communication *****
    clock_gettime(CLOCK_REALTIME, &h_start);

    if(p->threads == 1)
    {
      //when using one device just continue
      wAcc[p->device]++;

      clock_gettime(CLOCK_REALTIME, &h_end);
      comm_time[p->device] += getTime(h_start, h_end);

      continue;
    }

    //Transfer using chessboard ordering.
    //0<->1 2<->3 .. (n-2)<->(n-1) -----> 1<->2 ... (n-3)<->(n-2)
    if((p->device == 0) || (p->device == p->threads - 1))
    {
      switch(p->device)
      {
        case 0:
        {
          write_ghosts(0, 0, 1, i, p->grid + (p->h - p->border_size) * p->n,
            p->n * sizeof(float),
            dgrid + (p->h - p->border_size) * (pitch/sizeof(float)),
            pitch, p->n * sizeof(float), p->border_size);

          read_ghosts(0, 0, 1, i, p->grid + (p->h * p->n), p->n * sizeof(float),
            dgrid + p->h * (pitch/sizeof(float)), pitch, p->n * sizeof(float),
            p->border_size);

          break;
        }
        default:
        {
          write_ghosts(p->device, 2 * p->device - 1, 2 * p->device - 2, i,
            p->grid, p->n * sizeof(float),
            dgrid + (p->border_size) * (pitch/sizeof(float)),
            pitch, p->n * sizeof(float), p->border_size);

          read_ghosts(p->device, 2 * p->device - 1, 2 * p->device - 2, i,
            p->grid - offset, p->n * sizeof(float), dgrid, pitch,
            p->n * sizeof(float), p->border_size);
        }
      }
    } else
    {
      //if odd start from north otherwise from south
      if(p->device & 1)
      {
        write_ghosts(p->device, p->device * 2 - 1, 2 * (p->device - 1), i,
          p->grid, p->n * sizeof(float),
          dgrid + (p->border_size) * (pitch/sizeof(float)), pitch,
          p->n * sizeof(float), p->border_size);

        read_ghosts(p->device, p->device * 2 - 1, 2 * (p->device - 1), i,
          p->grid - (p->border_size * p->n), p->n * sizeof(float), dgrid,
          pitch, p->n * sizeof(float), p->border_size);

        write_ghosts(p->device, p->device * 2, p->device * 2 + 1, i,
          p->grid + (p->h - p->border_size) * p->n, p->n * sizeof(float),
          dgrid + p->h * (pitch/sizeof(float)), pitch, p->n * sizeof(float),
          p->border_size);

        read_ghosts(p->device, p->device * 2, p->device * 2 + 1, i,
          p->grid + (p->h * p->n), p->n * sizeof(float),
          dgrid + (p->h + p->border_size) * (pitch/sizeof(float)), pitch,
          p->n * sizeof(float), p->border_size);

      } else
      {
        write_ghosts(p->device, p->device * 2, p->device * 2 + 1, i,
          p->grid + (p->h - p->border_size) * p->n, p->n * sizeof(float),
          dgrid + p->h * (pitch/sizeof(float)), pitch, p->n * sizeof(float),
          p->border_size);

        read_ghosts(p->device, p->device * 2, p->device * 2 + 1, i,
          p->grid + (p->h * p->n), p->n * sizeof(float),
          dgrid + (p->h + p->border_size) * (pitch/sizeof(float)), pitch,
          p->n * sizeof(float), p->border_size);

        write_ghosts(p->device, p->device * 2 - 1, 2 * (p->device - 1), i,
          p->grid, p->n * sizeof(float),
          dgrid + p->border_size * (pitch/sizeof(float)), pitch,
          p->n * sizeof(float), p->border_size);

        read_ghosts(p->device, p->device * 2 - 1, 2 * (p->device - 1), i,
          p->grid - (p->border_size * p->n), p->n * sizeof(float), dgrid,
          pitch, p->n * sizeof(float), p->border_size);
      }
    }

    clock_gettime(CLOCK_REALTIME, &h_end);
    comm_time[p->device] += getTime(h_start, h_end);

//***** End Communication *****
  }

// CORE COMPUTATION - End ///////////////////////////

  //Last transfer. Just subdomain
  if(p->device == 0)
  {
    cudaEventRecord(start, 0);

    cudaMemcpy2D(p->grid, p->n * sizeof(float), dgrid, pitch,
      p->n * sizeof(float), p->h, cudaMemcpyDeviceToHost);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&temp_time, start, stop);
    ext_cpy_time[p->device] += temp_time;
  } else
  {
    cudaEventRecord(start, 0);

    cudaMemcpy2D(p->grid, p->n * sizeof(float),
      dgrid + p->border_size * (pitch/sizeof(float)), pitch, p->n * sizeof(float),
      p->h, cudaMemcpyDeviceToHost);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&temp_time, start, stop);
    ext_cpy_time[p->device] += temp_time;
  }

  //Format and display output
  if(p->threads == 1)
    b = p->iterations;
  printf("t%d avarage RSOR kernel time per iteration: %f ms.\n",
    p->device, r_kernel_time/p->iterations);
  printf("t%d avarage BSOR kernel time per iteration: %f ms.\n",
    p->device, b_kernel_time/p->iterations);
  printf("t%d total kernel time: %f ms.\n",
    p->device, r_kernel_time + b_kernel_time);
  printf("t%d MKT: %f ms.\n",
    p->device, (r_kernel_time + b_kernel_time) / p->iterations);

  printf("t%d total transfer time: %f ms [iter: %f + ext: %f].\n",
    p->device, cpy_time[p->device] + ext_cpy_time[p->device],
    cpy_time[p->device], ext_cpy_time[p->device]);
  printf("t%d MTT: %f ms.\n",
    p->device, cpy_time[p->device] / (p->iterations/b));
  printf("t%d total synchronization time: %f ms [iter: %f + ext: %f].\n",
    p->device, (sync_time[p->device] + ext_sync_time[p->device]) * 1E3,
    sync_time[p->device] * 1E3, ext_sync_time[p->device] * 1E3);
  printf("t%d MST: %f ms.\n",
    p->device, (sync_time[p->device] * 1E3) / (p->iterations/b));
  printf("t%d total communication time: %f ms [iter: %f + ext: %f].\n",
    p->device,
    comm_time[p->device] * 1E3
      + ext_sync_time[p->device] * 1E3 + ext_cpy_time[p->device],
    comm_time[p->device] * 1E3,
    ext_sync_time[p->device] * 1E3 + ext_cpy_time[p->device]);
  printf("t%d MCT: %f ms.\n",
    p->device, (comm_time[p->device] * 1E3) / (p->iterations/b));

  cudaEventDestroy(start);
  cudaEventDestroy(stop);

  cudaFree(dgrid);

  return NULL;
}

/*
 * call_solver
 *  solver - the index of the solver to use.
 *  grid - pointer to the PDE domain.
 *    n - width/height of the domain in terms of number of elements per row/column.
 *    iterations - the number of times the solver should iterate.
 *    threads - indicates how many threads, and finally devices, should cooperate
     in solving the PDE.
 *
 * Instantiates the right number of threads and manages their cooperation to solve
     the PDE.
 */

int call_solver(
  int solver, float *omega, float *grid, int m, int n,
  int iterations, int threads, int border_size
  )
{
  pthread_t *t_id = new pthread_t[threads];

  struct Params *t_p = new struct Params[threads];

  struct timespec start, end;

  mutex = new pthread_mutex_t[2 * (threads - 1)];
  rCond = new pthread_cond_t[2 * (threads - 1)];
  wCond = new pthread_cond_t[2 * (threads - 1)];
  rAcc = new int[2 * (threads - 1)];
  wAcc = new int[2 * (threads - 1)];

  ext_cpy_time = new double[threads];
  ext_sync_time = new double[threads];
  cpy_time = new double[threads];
  sync_time = new double[threads];
  comm_time = new double[threads];
  dev_sel_time = new double[threads];

  if(*omega == 0)
  {
    float pi = 4.f*atan(1.f);
    printf("pi: %.15f\n", pi);
    float interval = 1.f / (n + 1);
    *omega = 2.f / (1.f + sin(pi * interval));
    if(*omega > 1.9) *omega = 1.9;
  }

  for(int i = 0; i < 2 * (threads - 1); i++)
  {
    pthread_mutex_init(&mutex[i], NULL);
    pthread_cond_init(&rCond[i], NULL);
    pthread_cond_init(&wCond[i], NULL);
    rAcc[i] = 0;
    wAcc[i] = 0;
  }

  for(int i = 0; i < threads; i++)
  {
    t_p[i].solver = solver;
    t_p[i].omega = *omega;
    t_p[i].n = n;
    t_p[i].h = m/threads; //supposes threads%2 = 0
    t_p[i].border_size = border_size * (threads > 1);
    t_p[i].iterations = iterations;
    t_p[i].grid = grid + i * t_p[i].h * n;
    t_p[i].device = i;
    t_p[i].threads = threads;
  }

  if(t_p[0].h * threads < m)
    t_p[threads - 1].h += m - (t_p[0].h * threads);

  clock_gettime(CLOCK_REALTIME, &start);

  for(int i=0; i<threads; i++)
    pthread_create(&t_id[i], NULL, &tetra_thread, &t_p[i]);

  for(int i=0; i<threads; i++)
  {
    pthread_join(t_id[i], NULL);
  }

  clock_gettime(CLOCK_REALTIME, &end);

  double max_dev_sel = getMaxDeviceSelectionTime(threads);
  printf("Elapsed time: %.9fs\n", getTime(start, end) - max_dev_sel);

  for(int i = 0; i < 2 * (threads - 1); i++)
  {
    pthread_mutex_destroy(&mutex[i]);
    pthread_cond_destroy(&rCond[i]);
    pthread_cond_destroy(&wCond[i]);
  }

  delete [] mutex;
  delete [] rCond;
  delete [] wCond;
  delete [] rAcc;
  delete [] wAcc;

  delete [] t_id;
  delete [] t_p;

  delete [] ext_cpy_time;
  delete [] ext_sync_time;
  delete [] comm_time;
  delete [] sync_time;
  delete [] cpy_time;
  delete [] dev_sel_time;

  return 0;
}
```

B.2.4 Block Partitioning - Computation and Communication

Samples from *psync_blocks.cu*

```
/*
 * tetra_thread
 *  params - contains the parameters required to setup a partial solver (i.e. the
      solver running on one of the GPU and solving
 *  computing a specific portion of domain).
 */

void * tetra_thread(void *params)
{
  //Cast the cookie pointer to the right type
  struct Params *p = (struct Params *)params;

  //Vertical and horizontal neighbors
  int vn, hn;

  if(p->device % 2)
    hn = p->device - 1;
  else
    hn = p->device + 1;

  if(p->device <= 1)
    vn = p->device + 2;
  else
    vn = p->device - 2;

  struct timespec h_start, h_end;

  clock_gettime(CLOCK_REALTIME, &h_start);

  selectDevice(p->device);

  clock_gettime(CLOCK_REALTIME, &h_end);
  dev_sel_time[p->device] = getTime(h_start, h_end);

  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);

  float r_kernel_time = 0, b_kernel_time = 0, temp_time;

  //Setup subdomain processing
  float *dgrid;
  int offset, height;
  size_t pitch;

  //Total number of rows counting borders. GPUs in the middle
  //deal with two sides.
  height = p->h + (p->border_size);

  //Offset to the beginning of the subdomain
  switch(p->device)
  {
    case 0:
      offset = 0;
      break;
    case 1:
      offset = p->border_size;
      break;
    case 2:
      offset = p->border_size * p->stride;
      break;
    case 3:
      offset = p->border_size * p->stride + p->border_size;
  }
  //Allocate memory on the device
  cudaMallocPitch((void **)&dgrid, &pitch,
    (p->n + p->border_size) * sizeof(float), height);

  //Kernel Execution Configuration
  int grid_x, grid_y, local_w;

  //Number of threads per threadblock
  local_w = 112;

  dim3 blockDim(local_w);

  //First transfer. Entire subdomain+borders.
  comm_time[p->device] = ext_cpy_time[p->device] = cpy_time[p->device] = 0;

  cudaEventRecord(start, 0);

  cudaMemcpy2D(dgrid, pitch, p->grid - offset, p->stride * sizeof(float),
    (p->n + p->border_size) * sizeof(float), height, cudaMemcpyHostToDevice);

  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&temp_time, start, stop);
  ext_cpy_time[p->device] += temp_time;

  clock_gettime(CLOCK_REALTIME, &h_start);

  //Increase read counters.
  pthread_mutex_lock(&mutex[vn]);
    rAcc[vn]++;
  pthread_mutex_unlock(&mutex[vn]);
  pthread_cond_signal(&rCond[vn]);

  //Horizontal mutexes are in position i > 3
  pthread_mutex_lock(&mutex[hn + 4]);
    rAcc[hn + 4]++;
  pthread_mutex_unlock(&mutex[hn + 4]);
  pthread_cond_signal(&rCond[hn + 4]);

  clock_gettime(CLOCK_REALTIME, &h_end);
  ext_sync_time[p->device] = getTime(h_start, h_end);
  sync_time[p->device] = 0;
  comm_time[p->device] += sync_time[p->device];

// CORE COMPUTATION - Begin ///////////////////////////

  //So to be able to iterate with border_size == 0
  int b = p->border_size > 0 ? p->border_size : 1;

  for(int i = 0; i < p->iterations/b; i++)
  {
    int border = b;
    //k_* are offsets used to point to the beginning of the area to process
    int k_v = 0, k_h = 0;
    while(border > 0)
    {
      //Compute grid dimensions.
      //Horizontally, blocks shift with window size local_w-2.
      //Vertically, blocks shift with window size 2.
      grid_x = (p->n + border) / (local_w - 2);
      if(grid_x * (local_w - 2) < p->n + border) grid_x++;

      grid_y = (2 * height - 4) / sizeof(float) + (height % 2);

      dim3 gridDim(grid_x, grid_y);

      switch(p->solver)
      {
        case 0:
        {
```

```
      /* apply R/B SOR, using global mem for fetching data. */
      cudaEventRecord(start, 0);

      rsor_gm_block <<<gridDim, blockDim, 4 * local_w * sizeof(float)>>> (
        dgrid + k_v * (pitch/sizeof(float)) + k_h, p->n + border, height,
        p->omega, pitch/sizeof(float),
        (p->device % 2) * ((p->stride - p->n) - border),
        (p->device>1) * (p->h-border), p->device);

      cudaEventRecord(stop, 0);
      cudaEventSynchronize(stop);
      cudaEventElapsedTime(&temp_time, start, stop);
      r_kernel_time += temp_time;

      cudaEventRecord(start, 0);

      bsor_gm_block <<<gridDim, blockDim, 4 * local_w * sizeof(float)>>> (
        dgrid + k_v * (pitch/sizeof(float)) + k_h, p->n + border, height,
        p->omega, pitch/sizeof(float),
        (p->device % 2) * ((p->stride - p->n) - border),
        (p->device>1) * (p->h - border), p->device);

      cudaEventRecord(stop, 0);
      cudaEventSynchronize(stop);

      cudaEventElapsedTime(&temp_time, start, stop);
      b_kernel_time += temp_time;

      break;
    }
    case 1:
    {
      /* apply R/B SOR, using a texture to fetch data */
      size_t tex_offset = selectTextureToBind(p->device, i,
        dgrid + k_v * (pitch/sizeof(float)) + k_h, height * pitch);

      cudaEventRecord(start, 0);

      rsor_lintex_block <<<gridDim, blockDim, 4 * local_w * sizeof(float)>>> (
        dgrid + k_v * (pitch/sizeof(float)) + k_h, p->n + border, height,
        p->omega, pitch/sizeof(float), (p->device % 2) * (p->n - border),
        (p->device > 1) * (p->h - border), tex_offset/sizeof(float),
        p->device);

      cudaEventRecord(stop, 0);
      cudaEventSynchronize(stop);

      cudaEventElapsedTime(&temp_time, start, stop);
      r_kernel_time += temp_time;

      cudaEventRecord(start, 0);

      bsor_lintex_block <<<gridDim, blockDim, 4 * local_w * sizeof(float)>>> (
        dgrid + k_v * (pitch/sizeof(float)) + k_h, p->n + border, height,
        p->omega, pitch/sizeof(float),
        (p->device % 2) * (p->n - border), (p->device > 1) * (p->h - border),
        tex_offset/sizeof(float), p->device);

      cudaEventRecord(stop, 0);
      cudaEventSynchronize(stop);
      cudaEventElapsedTime(&temp_time, start, stop);
      b_kernel_time += temp_time;

      break;
    }
  }

  //Eventually increase offsets
  border--; height--;
  k_v += (p->device > 1); k_h += (p->device % 2);
```

```
    } //End border consuming while

    //Restore height
    height += b;

//***** Communication *****
#ifdef TIMER
    clock_gettime(CLOCK_REALTIME, &h_start);
#endif
    switch(p->device)
    {
      case 0:
        moveBlockToHost(0, 2, i, 0/*vertical*/,
          p->grid + (p->h - p->border_size) * p->stride,
          p->stride * sizeof(float),
          dgrid + (p->h - p->border_size) * pitch/sizeof(float), pitch,
          p->n * sizeof(float), p->border_size);
        moveBlockToDevice(0, 2, i, 0/*vertical*/, p->grid + p->h * p->stride,
          p->stride * sizeof(float), dgrid + (p->h) * pitch/sizeof(float),
          pitch, p->n * sizeof(float), p->border_size);

        moveBlockToHost(0, 1, i, 4/*horizontal*/, p->grid + (p->n - p->border_size)
            ,
          p->stride * sizeof(float), dgrid + (p->n - p->border_size),
          pitch, p->border_size * sizeof(float), p->h - p->border_size);
        moveBlockToDevice(0, 1, i, 4/*horizontal*/, p->grid + p->n,
          p->stride * sizeof(float), dgrid+(p->n), pitch,
          p->border_size * sizeof(float), p->h + p->border_size);
        break;
      case 1:
        moveBlockToHost(1, 3, i, 0/*vertical*/,
          p->grid + (p->h - p->border_size) * p->stride, p->stride * sizeof(float),
          dgrid + (p->h - p->border_size) * pitch/sizeof(float) + p->border_size,
          pitch, p->n * sizeof(float), p->border_size);
        moveBlockToDevice(1, 3, i, 0/*vertical*/, p->grid + p->h * p->stride,
          p->stride * sizeof(float),
          dgrid + (p->h) * pitch/sizeof(float) + p->border_size,
          pitch, p->n * sizeof(float), p->border_size);

        moveBlockToHost(1, 0, i, 4/*horizontal*/, p->grid,
          p->stride * sizeof(float), dgrid + p->border_size,
          pitch, p->border_size * sizeof(float), p->h - p->border_size);
        moveBlockToDevice(1, 0, i, 4/*horizontal*/, p->grid - p->border_size,
          p->stride * sizeof(float), dgrid, pitch,
          p->border_size * sizeof(float), p->h + p->border_size);
        break;
      case 2:
        moveBlockToHost(2, 0, i, 0/*vertical*/, p->grid,
          p->stride * sizeof(float), dgrid + p->border_size * pitch/sizeof(float),
          pitch, p->n * sizeof(float), p->border_size);
        moveBlockToDevice(2, 0, i, 0/*vertical*/, p->grid-p->border_size*p->stride,
            p->stride*sizeof(float), dgrid,
          pitch, p->n*sizeof(float), p->border_size);

        moveBlockToHost(2, 3, i, 4/*horizontal*/,
          p->grid + p->border_size * p->stride + p->n - p->border_size,
          p->stride * sizeof(float),
          dgrid + (2 * p->border_size) * pitch/sizeof(float) + p->n - p->
              border_size,
          pitch, p->border_size * sizeof(float), p->h - p->border_size);
        moveBlockToDevice(2, 3, i, 4/*horizontal*/,
          p->grid - (p->border_size * p->stride) + p->n,
          p->stride * sizeof(float), dgrid + p->n, pitch,
          p->border_size * sizeof(float), p->h + p->border_size);
        break;
      case 3:
        moveBlockToHost(3, 1, i, 0/*vertical*/, p->grid,
          p->stride * sizeof(float),
          dgrid + p->border_size * pitch/sizeof(float) + p->border_size,
          pitch, p->n * sizeof(float), p->border_size);
        moveBlockToDevice(3, 1, i, 0/*vertical*/,
          p->grid - p->border_size * p->stride, p->stride * sizeof(float),
          dgrid + p->border_size, pitch, p->n * sizeof(float), p->border_size);

        moveBlockToHost(3, 2, i, 4/*horizontal*/,
          p->grid + p->border_size * p->stride, p->stride * sizeof(float),
          dgrid + (2 * p->border_size) * pitch/sizeof(float) + p->border_size,
          pitch, p->border_size * sizeof(float), p->h - p->border_size);
        moveBlockToDevice(3, 2, i, 4/*horizontal*/,
          p->grid - (p->border_size * p->stride) - p->border_size,
          p->stride * sizeof(float), dgrid, pitch,
          p->border_size * sizeof(float), p->h + p->border_size);
        break;
    }

    clock_gettime(CLOCK_REALTIME, &h_end);
    comm_time[p->device] += getTime(h_start, h_end);

//***** End Communication *****
  }

// CORE COMPUTATION - End ///////////////////////////

  //Last transfer. Just subdomain
  switch(p->device)
  {
    case 0:
      offset = 0;
      break;
    case 1:
      offset = p->border_size;
      break;
    case 2:
      offset = p->border_size*pitch/sizeof(float);
      break;
    case 3:
      offset = p->border_size*pitch/sizeof(float)+p->border_size;
      break;
  }

    cudaEventRecord(start, 0);

    cudaMemcpy2D(p->grid, p->stride * sizeof(float), dgrid + offset, pitch,
      p->n * sizeof(float), p->h, cudaMemcpyDeviceToHost);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&temp_time, start, stop);
    ext_cpy_time[p->device] += temp_time;

  //Format and display output
  printf("t%d avarage RSOR kernel time: %f ms.\n",
    p->device, r_kernel_time / p->iterations);
  printf("t%d avarage BSOR kernel time: %f ms.\n",
    p->device, b_kernel_time / p->iterations);
  printf("t%d total kernel time: %f ms.\n",
    p->device, r_kernel_time + b_kernel_time);
  printf("t%d MKT: %f ms.\n",
    p->device, (r_kernel_time + b_kernel_time) / p->iterations);

  printf("t%d total transfer time: %f ms [iter: %f + ext: %f].\n",
    p->device, cpy_time[p->device] + ext_cpy_time[p->device],
    cpy_time[p->device], ext_cpy_time[p->device]);
  printf("t%d MTT: %f ms.\n",
    p->device, cpy_time[p->device] / (p->iterations/b));
  printf("t%d total synchronization time: %f ms [iter: %f + ext: %f].\n",
    p->device, (sync_time[p->device] + ext_sync_time[p->device]) * 1E3,
    sync_time[p->device] * 1E3, ext_sync_time[p->device] * 1E3);
  printf("t%d MST: %f ms.\n",
    p->device, (sync_time[p->device] * 1E3) / (p->iterations/b));
  printf("t%d total communication time: %f ms [iter: %f + ext: %f].\n", p->device,
    comm_time[p->device] * 1E3
```

```
      + ext_sync_time[p->device] * 1E3 + ext_cpy_time[p->device],
    comm_time[p->device] * 1E3,
    ext_sync_time[p->device] * 1E3 + ext_cpy_time[p->device]);
  printf("t%d MCT: %f ms.\n",
    p->device, (comm_time[p->device] * 1E3) / (p->iterations/b));

  cudaEventDestroy(start);
  cudaEventDestroy(stop);

  cudaFree(dgrid);

  return NULL;
}

/*
 * call_solver
 *  solver - the index of the solver to use.
 *  grid - pointer to the PDE domain.
 *    n - width/height of the domain in terms of number of elements per row/column.
 *    iterations - the number of times the solver should iterate.
 *    threads - indicates how many threads, and finally devices, should cooperate
     in solving the PDE.
 *
 *  Instantiates the right number of threads and manages their cooperation to solve
      the PDE.
 */

int call_solver(
  int solver, float *omega, float *grid, int m, int n,
  int iterations, int threads, int border_size
  )
{
  pthread_t *t_id = new pthread_t[threads];

  struct Params *t_p = new struct Params[threads];

  struct timespec start, end;

  mutex = new pthread_mutex_t[2 * threads];
  rCond = new pthread_cond_t[2 * threads];
  wCond = new pthread_cond_t[2 * threads];
  rAcc = new int[2 * threads];
  wAcc = new int[2 * threads];

  ext_cpy_time = new double[threads];
  ext_sync_time = new double[threads];
  cpy_time = new double[threads];
  sync_time = new double[threads];
  comm_time = new double[threads];
  dev_sel_time = new double[threads];

  if(*omega == 0)
  {
    float pi = 4.f * atan(1.f);
    float interval = 1.f / (n + 1);
    *omega = 2.f / (1.f + sin(pi * interval));
    if(*omega > 1.9) *omega = 1.9;
  }

  for(int i = 0; i< 2 * threads; i++)
  {
    pthread_mutex_init(&mutex[i], NULL);
    pthread_cond_init(&rCond[i], NULL);
    pthread_cond_init(&wCond[i], NULL);
    rAcc[i] = 0;
    wAcc[i] = 0;
  }

  for(int i = 0; i < threads; i++)
  {
    t_p[i].solver = solver;
    t_p[i].omega = *omega;
    t_p[i].stride = n;
    t_p[i].n = n/2;
    t_p[i].h = m/2;
    t_p[i].border_size = border_size;
    t_p[i].iterations = iterations;
    t_p[i].grid = grid + t_p[i].h * n * (i>1) + t_p[i].n * (i % 2 != 0);
    t_p[i].device = i;
    t_p[i].threads = threads;
  }

  if(t_p[0].h * threads < m)
    t_p[2].h = (t_p[3].h += m - (t_p[0].h * threads));

  if(t_p[0].n * threads < n)
    t_p[0].n = (t_p[1].n += n - (t_p[0].n * threads));

  clock_gettime(CLOCK_REALTIME, &start);

  for(int i=0; i<threads; i++)
    pthread_create(&t_id[i], NULL, &tetra_thread, &t_p[i]);

  for(int i=0; i<threads; i++)
  {
    pthread_join(t_id[i], NULL);
  }

  clock_gettime(CLOCK_REALTIME, &end);

  double max_dev_sel = getMaxDeviceSelectionTime(threads);
  printf("Elapsed time: %.9fs\n", getTime(start, end)-max_dev_sel);

  for(int i = 0; i < 2 * threads; i++)
  {
    pthread_mutex_destroy(&mutex[i]);
    pthread_cond_destroy(&rCond[i]);
    pthread_cond_destroy(&wCond[i]);
  }

  delete [] mutex;
  delete [] rCond;
  delete [] wCond;
  delete [] rAcc;
  delete [] wAcc;

  delete [] t_id;
  delete [] t_p;

  delete [] ext_cpy_time;
  delete [] ext_sync_time;
  delete [] cpy_time;
  delete [] sync_time;
  delete [] comm_time;
  delete [] dev_sel_time;

  return 0;
}
```

B.2.5 Borders Exchange

Samples from *psync_strips.cu*

```
/*
 * write_ghosts
 *  Copies a shared area from device to host.
 */

void write_ghosts(
  int device, int block, int neighbor, int step, float *host_addr,
  size_t host_pitch, float *dev_addr, size_t dev_pitch, int width_byte, int height
  )
{

  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);

  float temp_time;
  timespec h_start, h_end;
  clock_gettime(CLOCK_REALTIME, &h_start);

  //Lock mutex to check condition variable. Proceed just if the neighbor
  //read the border during the previous iteration
  pthread_mutex_lock(&mutex[block]);
    while(rAcc[block] != step + 1)
    {
      pthread_cond_wait(&rCond[block], &mutex[block]);
    }
  pthread_mutex_unlock(&mutex[block]);

  clock_gettime(CLOCK_REALTIME, &h_end);
  sync_time[device] += getTime(h_start, h_end);
  cudaEventRecord(start, 0);

  cudaMemcpy2D(host_addr, host_pitch, dev_addr, dev_pitch, width_byte,
    height, cudaMemcpyDeviceToHost);

  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop); //Block until the event is actually recorded

  cudaEventElapsedTime(&temp_time, start, stop);
  cpy_time[device] += temp_time;

  clock_gettime(CLOCK_REALTIME, &h_start);

  //Lock mutex to increase the write counter.
  pthread_mutex_lock(&mutex[block]);
    wAcc[block]++;
  pthread_mutex_unlock(&mutex[block]);
  pthread_cond_signal(&wCond[block]);

  clock_gettime(CLOCK_REALTIME, &h_end);
  sync_time[device] += getTime(h_start, h_end);

  cudaEventDestroy(start);
  cudaEventDestroy(stop);
}

/*
 * read_ghosts
 *   Copies a shared area from host to device.
 */

void read_ghosts(
  int device, int block, int neighbor, int step, float *host_addr,
  size_t host_pitch, float *dev_addr, size_t dev_pitch, int width_byte, int height
  )
{
  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);

  float temp_time;
  timespec h_start, h_end;
  clock_gettime(CLOCK_REALTIME, &h_start);

  //Lock mutex to check condition variable. Proceed just if the neighbor
  //wrote the border during the previous iteration
  pthread_mutex_lock(&mutex[neighbor]);
```

```
    while(wAcc[neighbor] != step+1)
    {
      pthread_cond_wait(&wCond[neighbor], &mutex[neighbor]);
    }
  pthread_mutex_unlock(&mutex[neighbor]);

  clock_gettime(CLOCK_REALTIME, &h_end);
  sync_time[device] += getTime(h_start, h_end);
  cudaEventRecord(start, 0);

  cudaMemcpy2D(dev_addr, dev_pitch, host_addr, host_pitch, width_byte,
    height, cudaMemcpyHostToDevice);

  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);

  cudaEventElapsedTime(&temp_time, start, stop);
  cpy_time[device] += temp_time;

  clock_gettime(CLOCK_REALTIME, &h_start);

  //Lock mutex to increase the read counter.
  pthread_mutex_lock(&mutex[neighbor]);
    rAcc[neighbor]++;
  pthread_mutex_unlock(&mutex[neighbor]);
  pthread_cond_signal(&rCond[neighbor]);

  clock_gettime(CLOCK_REALTIME, &h_end);
  sync_time[device] += getTime(h_start, h_end);

  cudaEventDestroy(start);
  cudaEventDestroy(stop);
}
```

B.2.6 Strip Partitioning - Texture-based Red Kernel

Samples from *pde_kernels.cu*

```
#define ATD(x,y) (pitch)*(gridPosBase_j+(y)) + (gridPosBase_i+(x))
#define ATS(x,y) (local_w)*(y) + (x)

texture<float, 1, cudaReadModeElementType> texRef_0;
texture<float, 1, cudaReadModeElementType> texRef_1;
texture<float, 1, cudaReadModeElementType> texRef_2;
texture<float, 1, cudaReadModeElementType> texRef_3;

float __device__ selectTex1D(int gpu, int pos)
{
  switch(gpu)
  {
    case 0:
      return tex1Dfetch(texRef_0, pos);
    case 1:
      return tex1Dfetch(texRef_1, pos);
    case 2:
      return tex1Dfetch(texRef_2, pos);
    case 3:
      return tex1Dfetch(texRef_3, pos);
  }

  return 0.f;
}

/*
 * Red SOR through linear texturing.
 */

__global__ void rsor_lintex(
  float *dgrid, int n, int h, float omega, int pitch, int abs_start,
  size_t offset, int dev
  )
{
  //vertical window size = 2
  int gridPosBase_j = blockIdx.y * 2;
  //horizontal window size = blockDim.x - 2
  int gridPosBase_i = blockIdx.x * (blockDim.x - 2);

  int i = threadIdx.x;
  int local_w = blockDim.x;

  extern __shared__ float sgrid[];

  //If block can't use more than 3 rows
  int ok3 = gridPosBase_j + 3 < h;
  //Red first
  int r1 = (abs_start + gridPosBase_j + 1) & 1;
  //Traversing sign
  int sign = r1 + __mul24(1 - r1, -1);
  //Computation indexes
  int com_i = i + 1;
  int com_j = (2 - r1) + __mul24(sign, 1 & i);

  if(gridPosBase_i+i < n)
  {
    sgrid[ATS(i,0)] = selectTex1D(dev, ATD(i,0) + offset);
    sgrid[ATS(i,1)] = selectTex1D(dev, ATD(i,1) + offset);
    sgrid[ATS(i,2)] = selectTex1D(dev, ATD(i,2) + offset);
    if(ok3)
      sgrid[ATS(i,3)] = selectTex1D(dev, ATD(i,3) + offset);
  }

  __syncthreads();

  if((i<local_w-2)&&(gridPosBase_i+i<n-2))
  {
    if(ok3)
      sgrid[ATS(com_i, com_j)] +=
        omega * (
        (sgrid[ATS(com_i - 1, com_j)] + sgrid[ATS(com_i + 1, com_j)]
          + sgrid[ATS(com_i, com_j - 1)] + sgrid[ATS(com_i, com_j + 1)]) / 4.f
          - sgrid[ATS(com_i, com_j)]
        );
    else if(com_j != 2)
      sgrid[ATS(com_i, com_j)] +=
        omega * (
        (sgrid[ATS(com_i - 1, com_j)] + sgrid[ATS(com_i + 1, com_j)]
          + sgrid[ATS(com_i, com_j - 1)] + sgrid[ATS(com_i, com_j + 1)]) / 4.f
          - sgrid[ATS(com_i, com_j)]
        );
  }

  __syncthreads();

  if((i < local_w - 2) && (gridPosBase_i + i < n - 2))
  {
    if(ok3)
      dgrid[ATD(com_i, com_j)] = sgrid[ATS(com_i, com_j)];
    else if(com_j == 1)
      dgrid[ATD(com_i, 1)] = sgrid[ATS(com_i, 1)];
  }

}

/*
 * Black SOR through linear texturing.
 */

__global__ void bsor_lintex(
  float *dgrid, int n, int h, float omega, int pitch, int abs_start, int dev
  )
{
  //vertical window size = 2
  int gridPosBase_j = blockIdx.y * 2;
  //horizontal window size = blockDim.x - 2
  int gridPosBase_i = blockIdx.x * (blockDim.x - 2);

  int i = threadIdx.x;
  int local_w = blockDim.x;

  extern __shared__ float sgrid[];

  int ok3 = gridPosBase_j + 3 < h;
  //Red first
  int r1 = (abs_start + gridPosBase_j + 1) & 1;
  //Traversing sign
  int sign = __mul24(-1, r1) + (1 - r1);
  //Computation indexes
  int com_i = i + 1;
  int com_j = (1 + r1) + __mul24(sign, 1 & i);

  if(gridPosBase_i + i < n)
  {
    sgrid[ATS(i,0)] = selectTex1D(dev, ATD(i,0));
    sgrid[ATS(i,1)] = selectTex1D(dev, ATD(i,1));
    sgrid[ATS(i,2)] = selectTex1D(dev, ATD(i,2));
    if(ok3)
      sgrid[ATS(i,3)] = selectTex1D(dev, ATD(i,3));
  }

  __syncthreads();

  if((i < local_w - 2) && (gridPosBase_i + i < n - 2))
  {
    if(ok3)
      sgrid[ATS(com_i, com_j)] +=
        omega * (
        (sgrid[ATS(com_i - 1, com_j)] + sgrid[ATS(com_i + 1, com_j)]
          + sgrid[ATS(com_i, com_j - 1)] + sgrid[ATS(com_i, com_j + 1)]) / 4.f
          - sgrid[ATS(com_i, com_j)]
        );
    else if(com_j != 2)
      sgrid[ATS(com_i, com_j)] +=
        omega * (
        (sgrid[ATS(com_i - 1, com_j)] + sgrid[ATS(com_i + 1, com_j)]
          + sgrid[ATS(com_i, com_j - 1)] + sgrid[ATS(com_i, com_j + 1)]) / 4.f
          - sgrid[ATS(com_i, com_j)]
        );
  }

  __syncthreads();

  if((i < local_w - 2) && (gridPosBase_i + i < n - 2))
  {
    if(ok3)
      dgrid[ATD(com_i, com_j)] = sgrid[ATS(com_i, com_j)];
    else if(com_j == 1)
        dgrid[ATD(com_i,1)] = sgrid[ATS(com_i,1)];
  }

}
```

Appendix C

Benchmark Software Installation

The benchmark sources are collected in two folders *src* and *include*. The folder *bin* contains all the scripts and is the default destination folder for all the executables created using the the supplied Makefile. The Makefile presents the options shown in Table **??**.

Table C.1 – Makefile options.

psync_strips	Generates a PDE solver based on strip partitioning called *run_tetra_psync_strips*.
psync_blocks	Generates a PDE solver based on block partitioning called *run_tetra_psync_blocks*.
grid_gen	Generates the timer used for regression analysis described in Appendix D.2.
timer_reg	Generates the timer used for regression analysis described in Appendix D.2.
all	Generates all the aboves.
psync_strips_emu	Generates a PDE solver based on strip partitioning in CUDA emulation mode. The application is called *psync_strips_emu*.
psync_blocks_emu	Generates a PDE solver based on block partitioning in CUDA emulation mode. The application is called *psync_blocks_emu*.
ptxas	Shows the number of registers required by the kernels as well as local, shared, and constant memory usage.
ptx	Generates an assemply source file for the pde kernels.
clean	Removes all the generated files within the *bin* folder.

Appendix D

Helper Tools

In this appendix we show the code of some tools that we implemented to support us automating the most repetitive and long phases of the benchmarking activity.

D.1 Domain Generator

D.1.1 Usage

The tool takes in input three parameters:

M The number of rows of the domain;

N The number of columns of the domain;

Filename The domain's output file.

So the command

```
$> ./grid_gen -m 1024 -n 1024 -f mib.in
```

creates a domain of dimensions 1024×1024 in *mib.in* with border's elements set to *MAX_TEMP*.

D.1.2 Code: *grid_gen.c*

```
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>

#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

#include <sys/time.h>
#include <sys/resource.h>

#include <errno.h>

#define MAX_TEMP 100
#define K_1GiB 1024*1024*1024.l

//command-line options (short version)
```

```
const char * const short_options = "ht:m:n:f:";

//command-line options (long version)
const struct option long_options[] = {
  {"help",        0,  NULL, 'h'},
  {"rows",        1,  NULL, 'm'},
  {"columns",     1,  NULL, 'n'},
  {"file",        1,  NULL, 'f'}
};

/*
 * Print usage information to <stream> and exit with <exit_code>.
 */
void print_usage(FILE* stream, int exit_code)
{
  fprintf(stream, "Usage: grid_gen -m <rows> -n <columns> -f <file> [-h]\n");

  exit(exit_code);
}

/*
 * MAIN FUNCTION
 */
int main(int argc, char **argv)
{
  size_t m, n;
  const char* outfile = NULL;
  int next_opt, opt_counter = 0;

  do{
    next_opt = getopt_long(argc, argv, short_options, long_options, NULL);

    switch(next_opt)
    {
      case 'h':
        print_usage(stdout, 0);
      case 'm':
       m = atoi(optarg);
        opt_counter++;
        break;
      case 'n':
        n = atoi(optarg);
        opt_counter++;
        break;
      case 'f':
        outfile = optarg;
        opt_counter++;
        break;
      case '?':
        fprintf(stderr, "Unexpected option.\n");
        print_usage(stderr, 1);
      case -1:
        //Done with options
        break;
      default:
        exit(-1);
    }
  } while(next_opt != -1);

  if(opt_counter != 3)
  {
    fprintf(stderr, "Missing mandatory options.\n");
    print_usage(stderr, 1);
  }

  size_t size = m*n;
  size_t iter = (size*sizeof(float))/(K_1GiB);
  size_t rest = size*sizeof(float)-iter*(K_1GiB);

  int fd;
  if((fd = open(outfile,
    O_CREAT | O_RDWR | O_TRUNC,  S_IRUSR | S_IWUSR | S_IRGRP)) == -1)
  {
    fprintf(stderr, "Error opening file.\n");
    return EXIT_FAILURE;
  }

  float *grid;

  grid = (float *)calloc(size, sizeof(float));

  for(size_t j=0; j<m; j++)
  {
    grid[j*n] = MAX_TEMP;
    grid[j*n+n-1] = MAX_TEMP;
  }
  for(size_t i=0; i<n; i++)
  {
    grid[i] = MAX_TEMP;
    grid[(m-1)*n+i] = MAX_TEMP;
  }

  ssize_t r;
  size_t total = 0;
  for(size_t i = 0; i < iter; i++)
  {
    if((r=pwrite(fd, grid, K_1GiB, i*K_1GiB)) != K_1GiB)
    {
      fprintf(stderr, "At iteration %ld pwrite returns %ld.\n", i, r);
      perror(NULL);
      exite(EXIT_FAILURE);
    }
    total += r;
  }

  if((r=pwrite(fd, grid, rest, total)) != rest)
  {
    fprintf(stderr, "Writing rest pwrite returns %ld.\n", r);
    perror(NULL);
    exite(EXIT_FAILURE);
  }
  total += r;

  struct stat fs;

  fstat(fd, &fs);

  if(fs.st_size != size*sizeof(float))
  {
    printf("Error: size differs from what expected.
        Expected: %ld, Size: %ld, # allocated blocks: %ld\n",
      size*sizeof(float), fs.st_size, fs.st_blocks);
    exit(EXIT_FAILURE);
  }

  printf("done. Size: %ld, # allocated blocks: %ld\n",
    fs.st_size, fs.st_blocks);
  printf("float size: %ld.\n", sizeof(float));

  close(fd);
  free(grid);

  exit(EXIT_SUCCESS);
}
```

D.2 Timer

D.2.1 Usage

The tool takes in input an integer parameter N and performs some basic timing experiments based on a $N \times N$ matrix of floats (e.g. retrieving *cudaMemcpy2D()* startup time). It was used during the regression analysis in Section 6.3.

D.2.2 Code: ***timer_reg.cu***

```
#include <stdio.h>

int main(int argc, char **argv)
{
  cudaEvent_t start, stop;
  float time;

  cudaEventCreate(&start);
  cudaEventCreate(&stop);

  int n = atoi(argv[1]);

  size_t pitch;

  float *dgrid;
  float *hgrid = (float *)malloc(n * n * sizeof(float));
  cudaMallocPitch((void **)&dgrid, &pitch, n * sizeof(float), n);

  //Compute the time required to send 0Byte (Startup)
  float startup;
  cudaEventRecord(start, 0);

  cudaMemcpy2D(dgrid, pitch, hgrid, n * sizeof(float),
    0, 0, cudaMemcpyHostToDevice);

  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&startup, start, stop);
  printf("CudaMemcpy2D startup: %fms\n", startup);

  //Compute the time required to send 1Byte
  float mean = 0.f;
  for(int i=0; i<1000; i++)
  {
    cudaEventRecord(start, 0);

    cudaMemcpy2D(dgrid, pitch, hgrid, n * sizeof(float),
      1, 1, cudaMemcpyHostToDevice);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);
    mean += time;
  }
  float tword = mean/1000-startup;
  printf("CudaMemcpy2D 1 Byte: %fms\n", tword);

  //Time required to transfer n/2*n elements
  cudaEventRecord(start, 0);

  cudaMemcpy2D(dgrid, pitch, hgrid, n * sizeof(float), n * sizeof(float),
    n/2, cudaMemcpyHostToDevice);

  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&time, start, stop);
  printf("CudaMemcpy2D to send %d x %d/2: %fms\n", n, n, time);

  //Time required to transfer n*n/2 elements
  cudaEventRecord(start, 0);

  cudaMemcpy2D(dgrid, pitch, hgrid + n * n/2, n * sizeof(float),
    n * sizeof(float)/2, n, cudaMemcpyHostToDevice);

  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&time, start, stop);
  printf("CudaMemcpy2D to send %d x %d/2: %fms\n", n, n, time);

  //Time required to transfer n  elements
  cudaEventRecord(start, 0);

  cudaMemcpy2D(dgrid, pitch, hgrid, n * sizeof(float), n * sizeof(float),
    n, cudaMemcpyHostToDevice);

  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&time, start, stop);
  printf("CudaMemcpy2D to send %d x %d: %fms\n", n, n, time);

  cudaFree(dgrid);
  cudaEventDestroy(start);
  cudaEventDestroy(stop);

  free(hgrid);

  return 0;
}
```

D.3 Domains Populator

D.3.1 Usage

The tool is a script with a configuration area. It populates a set of domains based on a list of dimensions and a list of shapes. For example, configuring the script with the two lists:

```
sizes = [128]
shapes=[[0.5,1],[1,1]]
```

it would generate two domains with dimensions respectively 64×128 and 128×128.

D.3.2 Code: *populate.py*

```
#!/usr/bin/python
#——————————————————————
# Config section
#——————————————————————

#Define the directory containing the programs to be run (relative to this script)
program="./bin/grid_gen"
#Define the directory containing the input files (relative to this script)
inputDir="./in/"

#Define the sizes
#e.g. sizes = [14000, 33000]

#Define the shapes as dimensional factors:
#With size s and factors [f,g] -> [s*f, s*g] matrix
```

```
#e.g. shapes=[[0.5,1],[1,1],[1,0.5]]

#------------------------------------------------------------
# Do not edit beyond this line
#------------------------------------------------------------
from subprocess import *

##############################################################

def main():

    total = 0;
    for shape in shapes:
        for size in sizes:
            m = int(shape[0]*size)
            n = int(shape[1]*size)
            size = m*n*4
            cmdLine = program + " " + str(m) + " " + str(n) + " " + inputDir
              + str(m) + "." + str(n) + ".in"
            if size < 1E3:
               print "Execution of: " + cmdLine
                + "\tGenerating " + str(size) + " bytes..."
            elif size >= 1E3 and size < 1E6:
               print "Execution of: " + cmdLine
                + "\tGenerating %.2f KB..." %(size/1E3)
            elif size >= 1E6 and size < 1E9:
               print "Execution of: " + cmdLine
                + "\tGenerating %.2f MB..." %(size/1E6)
            else:
               print "Execution of: " + cmdLine
                + "\tGenerating %.2f GB..." %(size/1E9)
            p = Popen(cmdLine, shell=True, stdin=PIPE,
              stdout=PIPE, close_fds=True)
            p.wait()
            if p.returncode != 0:
               print "Something went wrong launching grid_gen.
                Return code " + str(p.returncode)
            else:
               print "Done"
               total += size
    if total < 1E3:
       print "Created " + str(total) + " bytes of good data."
    elif total >= 1E3 and total < 1E6:
       print "Created %.2f KB of good data." %(total/1E3)
    elif total >= 1E6 and total < 1E9:
       print "Created %.2f MB of good data." %(total/1E6)
    else:
       print "Created %.2f GB of good data." %(total/1E9)

#----------------------------------------------------------------
#Execution starts here
#----------------------------------------------------------------
if __name__=="__main__":
    main()
```

D.4 Benchmark Launcher

D.4.1 Usage

The tool is a script used to launch several different configurations of the *run_tetra* benchmark application. In the configuration area, it allows to specify:

- Different versions of the benchmark;
- A dictionary with entries of the form: $< \#GPUs, [listofsizes] >$;

- A dictionary with entries of the form: $< \#GPUs, [listofshapes] >$;
- A dictionary with entries of the form: $< Benchmarkversion, [\#GPUs] >$;
- A list of border widths;
- The relaxation factor omega;
- Whether to save output images or not.

The script launches the different configurations and collects output results in a *cvs* file with columns' format:

M	N	Border_Size	MKT[ms]	MST[ms]	MTT[ms]	MCT[ms]	Elapsed Time[s]

D.4.2 Code: *launcher.py*

```
#!/usr/bin/python
#-----------------------------
# Config section
#-----------------------------

#Define the directory containing the programs to be run (relative to this script)
programsDir="./bin/"
#Define the directory containing the input files (relative to this script)
inputDir="./in/"
#Define the directory containing the input files (relative to this script)
outputDir="./out/"
#Set the list of programs to be run
programs=["run_tetra_psync_strips","run_tetra_psync_blocks"]

#Define the sizes
# sizes = {1: [128, 200], 2: [128, 200, 256], 4: [128]}

#Define the shapes as dimensional factors:
#With size s and factors [f,g] -> [s*f, s*g] matrix
#shapes = {1: [[1,1]], 2: [[1,1]], 4: [[0.5,1],[1,1],[1,0.5]]}

#Define the number of GPUs to use relatively to the program index
#gpusList = {0: [1,2,4], 1: [4]}

#Define the number of GPUs to use relatively to the program index
bsList = [1, 2, 4, 5, 8, 10, 20, 25, 40, 50, 100]

#Define the relaxation factor omega (0 = use the optimum value)
omega=0
#Save graphics output
img=0

#------------------------------------------
# Do not edit beyond this line
#------------------------------------------
from subprocess import *
import re
import sys
import math
import time

###############################################

def setupOutFile(outFileName):

    outFile=file(outputDir+outFileName,"w")
    outFile.write("M;N;Border_Size;MKT[ms];MST[ms];
      MTT[ms];MCT[ms];Elapsed Time[s]\n")
    return outFile

##################################################

def runTetra(program, gpus, m, n, bs, k):

    #Prepare the extraction regexp
    mktREs=[]
    mstREs=[]
    mttREs=[]
    mctREs=[]
    for gpu in range(gpus):
        mktREs.append(re.compile("t"+str(gpu)+" MKT: (\d+.\d+)"))
        mstREs.append(re.compile("t"+str(gpu)+" MST: (\d+.\d+)"))
        mttREs.append(re.compile("t"+str(gpu)+" MTT: (\d+.\d+)"))
        mctREs.append(re.compile("t"+str(gpu)+" MCT: (\d+.\d+)"))
    elapsedRE=re.compile("Elapsed time: (\d+.\d+)")

    dim = n;
    if (dim < m): dim = m;

    int_iter=100
    if (dim >= 33000):
       ext_iter=5
    elif (dim >= 8000) and (n < 33000):
       ext_iter=10
    elif (dim >= 4000) and (n < 8000):
       ext_iter=20
    elif (dim >= 128) and (n < 4000):
       ext_iter=30

    resList=[]
    cmdLine=programsDir + program + " " + str(gpus) + " " + str(m) + " "
      + str(n) + " " +inputDir+str(m)+"."+str(n)+".in"+ " " + str(int_iter)
      + " " + str(bs) + " " + str(k) + " " + str(omega) + " " + str(img)

    ttime=0

    for i in range(ext_iter):
        mkts = []
        msts = []
        mtts = []
        mcts = []
        start = time.time()
        p = Popen(cmdLine, shell=True, stdin=PIPE,
          stdout=PIPE, close_fds=True)
        p.wait()
        output = p.stdout.read()
        elapsed = time.time()-start
        if p.returncode != 0:
           print "Run #"+str(i)+" - Error. Return code " + str(p.returncode)
           exit(1)
        else:
           if elapsed > 60:
              print "Run #%d - Execution went fine in %.2fmin.
                Extracting relevant data..." % (i, elapsed/60)
           else:
              print "Run #%d - Execution went fine in %.2fs.
                Extracting relevant data..." % (i, elapsed)
        ttime += elapsed
        #Extract
        for gpu in range(gpus):
            mkts.append(float(mktREs[gpu].search(output).group(1)))
            msts.append(float(mstREs[gpu].search(output).group(1)))
            mtts.append(float(mttREs[gpu].search(output).group(1)))
            mcts.append(float(mctREs[gpu].search(output).group(1)))
        resList.append([]);
        resList[i].append(max(mkts))
        resList[i].append(max(msts))
        resList[i].append(max(mtts))
```

```
            resList[i].append(max(mcts))
            resList[i].append(float(elapsedRE.search(output).group(1)))
    final=ttime/60
    if final > 60:
        print "Done %d runs in %.2fh. Saving result..." % (ext_iter, final/60)
    else:
        print "Done %d runs in %.2fmin. Saving result..." % (ext_iter, ttime/60)

    return [ext_iter, int_iter, resList]

##################################################

def main():
    if len(sys.argv) < 2 or len(sys.argv) > 3:
        print "Usage 1: launcher <border_size>"
        print "Usage 2: launcher <border_size_inf> <border_size_sup>"
        exit(1)
    elif len(sys.argv) == 2:
          sup = inf = int(sys.argv[1])
    else:
          inf = int(sys.argv[1])
          sup = int(sys.argv[2])

    if inf < 0 or sup < inf:
        print "Need 0 <= inf <= sup"
        exit(1)

    for p in [0,1]:
        for k in [0,1]:
            for gpus in gpusList[p]:
                for shape in shapes[gpus]:
                    print "Preparing the output file..."
                    outFile=setupOutFile("res_" + programs[p] + "_k" + str(k)
                      + "_g" + str(gpus) + "_s"
                      + str(shapes[gpus].index(shape) + 1) + ".csv")
                    print "Done"
                    for size in sizes[gpus]:
                        for bs in bsList:
                            m = int(shape[0]*size)
                            n = int(shape[1]*size)
                            print "Execution of: " + programs[p] + "[GPUs="
                              + str(gpus) + ",M=" + str(m) + ",N=" + str(n)
                              + ",K="+ str(k) + ",bs="+ str(bs) + ",omega="
                              + str(omega) + ",saveImg=" + str(img) +"]"

                            result=runTetra(programs[p], gpus, m, n, bs, k)
                            for res in result[2]:
                                outFile.write(str(m) + "\t" + str(n) + "\t"
                                  + str(bs) + "\t" + str(res[0]) + "\t"
                                  + str(res[1]) + "\t" + str(res[2]) + "\t"
                                  + str(res[3]) + "\t" + str(res[4]) + "\n")
                                outFile.flush()
                            print "Saved."
                    outFile.close()
    print "Stop"

#--------------------------------------------------
#Execution starts here
#--------------------------------------------------
if __name__=="__main__":
    main()
```

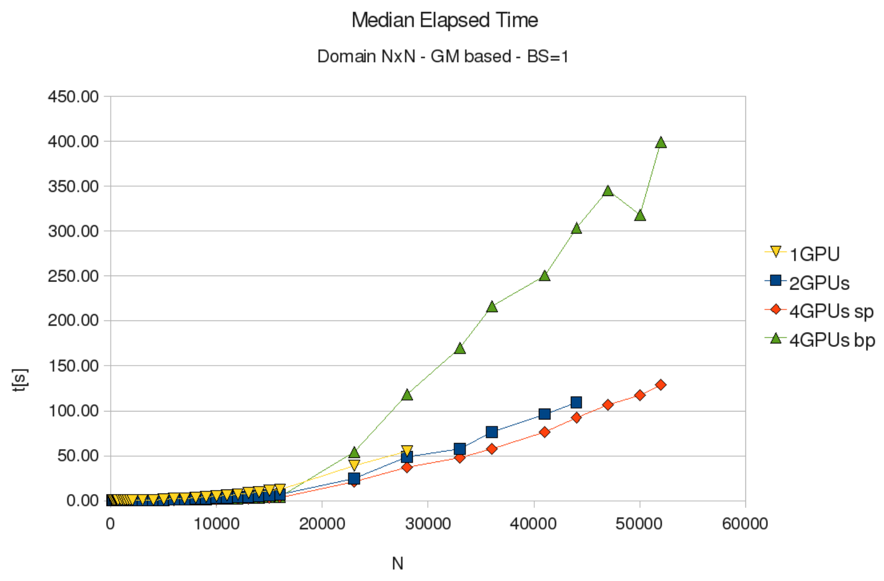
Appendix E

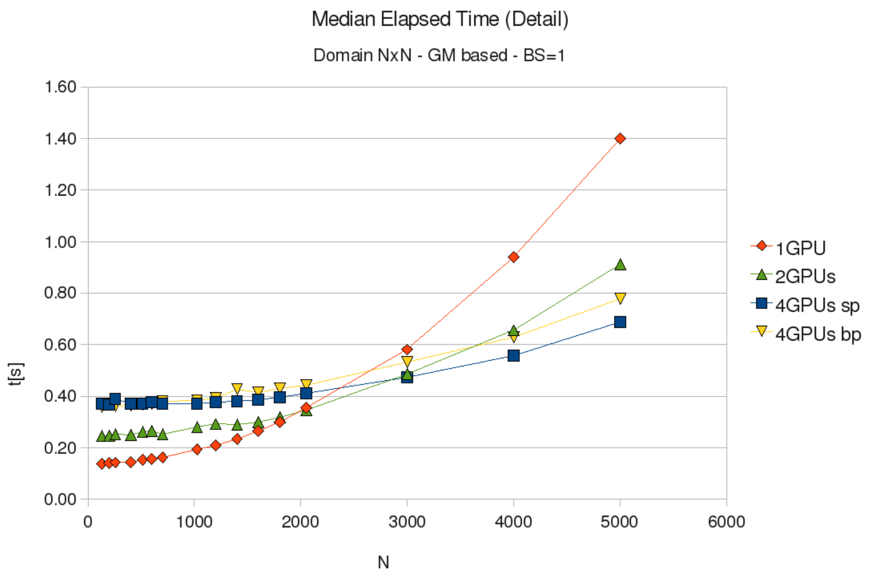
Graphs Collection

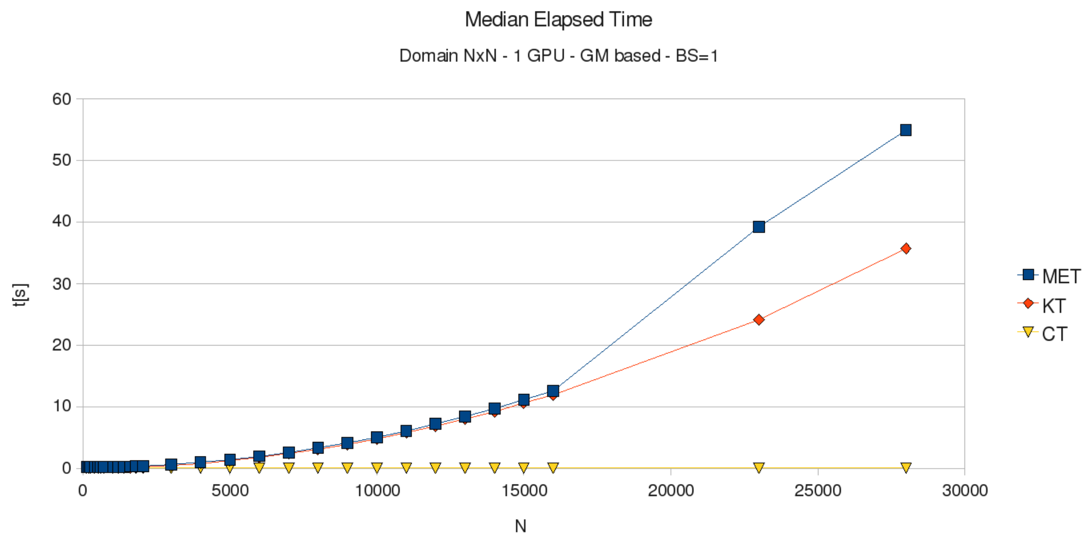
In this appendix we show many of the graphs obtained analyzing the results discussed in Chapter 6. We have collect them depending on the execution round. Rounds are mainly characterized by the domain size and the border width. In the first two rounds we used unitary border width and different shapes. In the last round we used squared domains with varying border sizes.

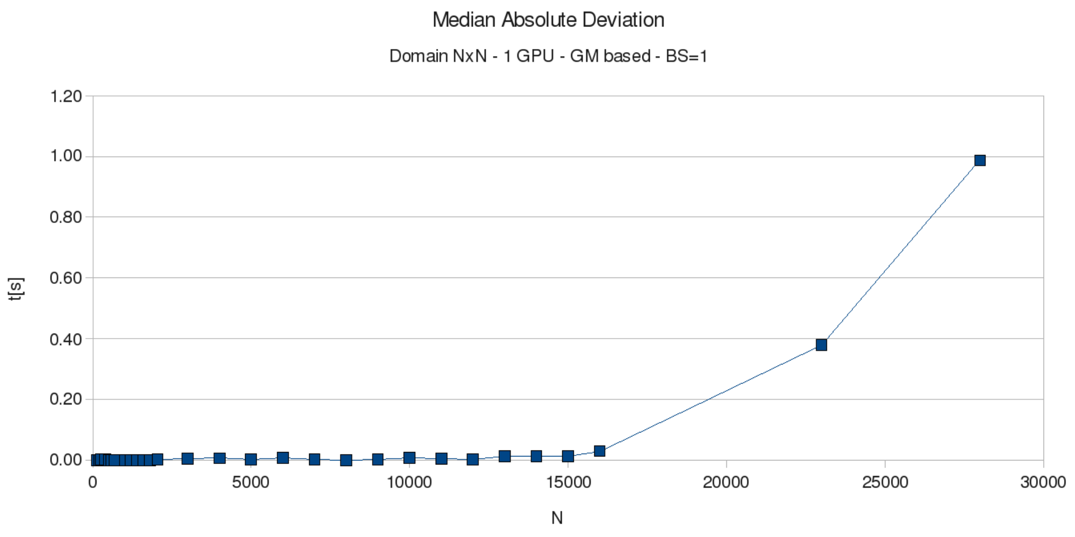
E.1 Execution Round 1

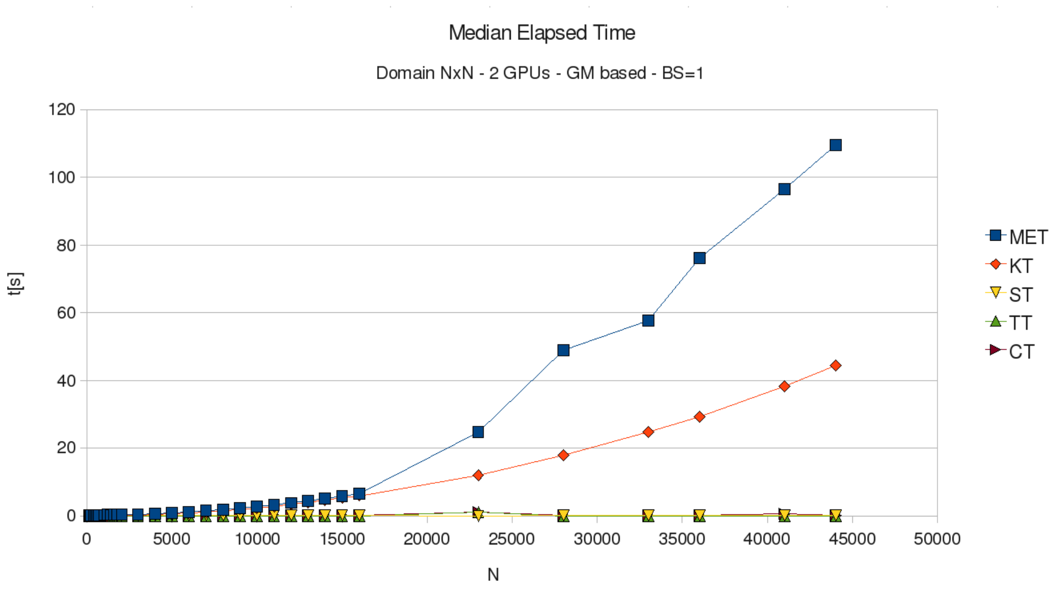
E.1.1 Global Memory-Based Kernels

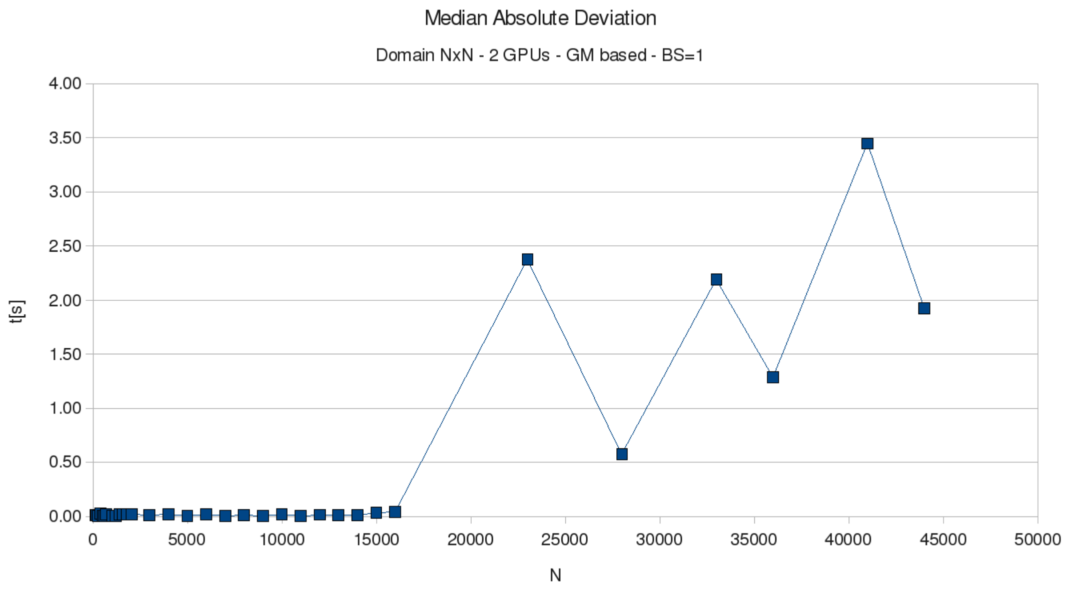


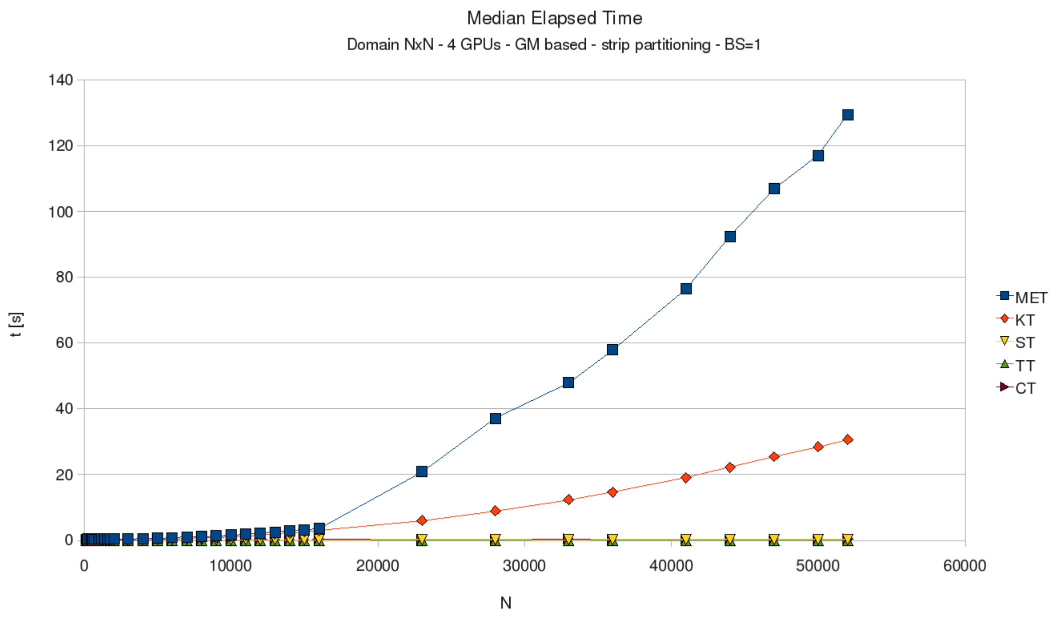

Median Elapsed Time (Detail)
Domain NxN - GM based - BS=1
t[s]
N
1GPU
2GPUs
4GPUs sp
4GPUs bp

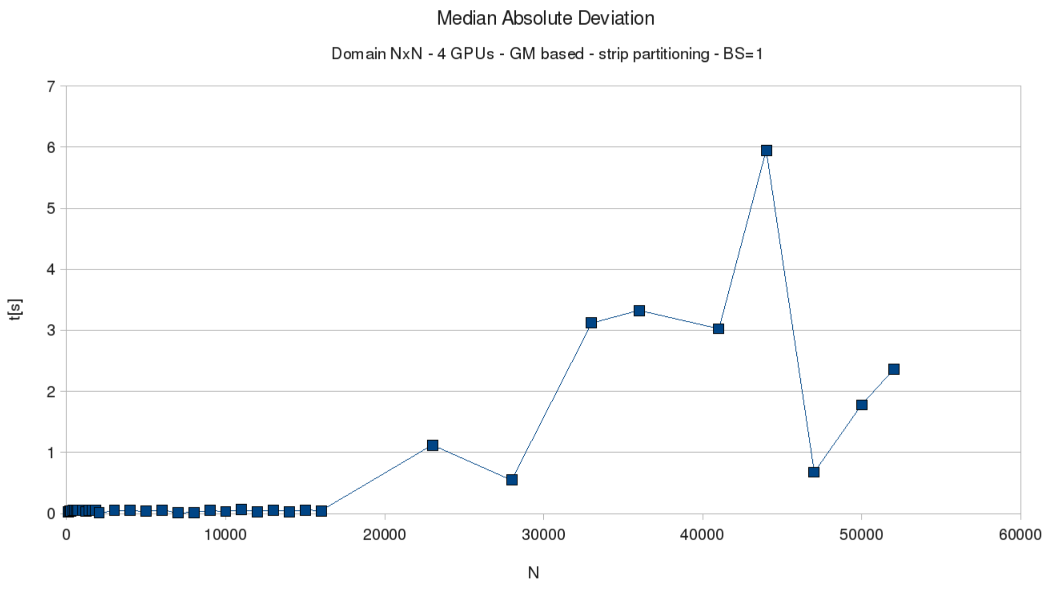

Median Elapsed Time
Domain NxN - 1 GPU - GM based - BS=1
t[s]
N
MET
KT
CT

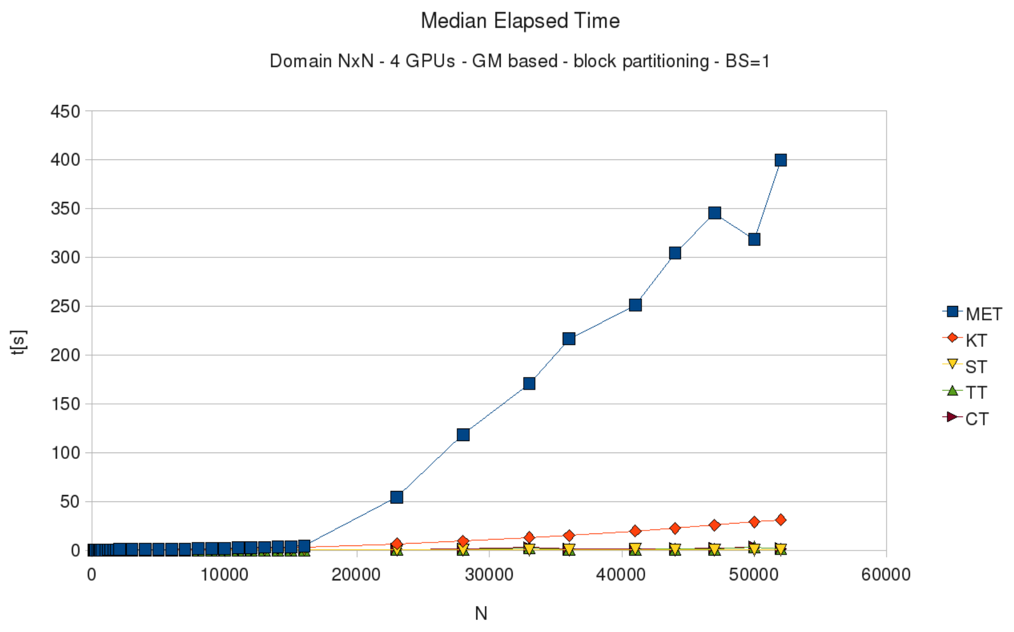

Median Absolute Deviation
Domain NxN - 1 GPU - GM based - BS=1
t[s]
N

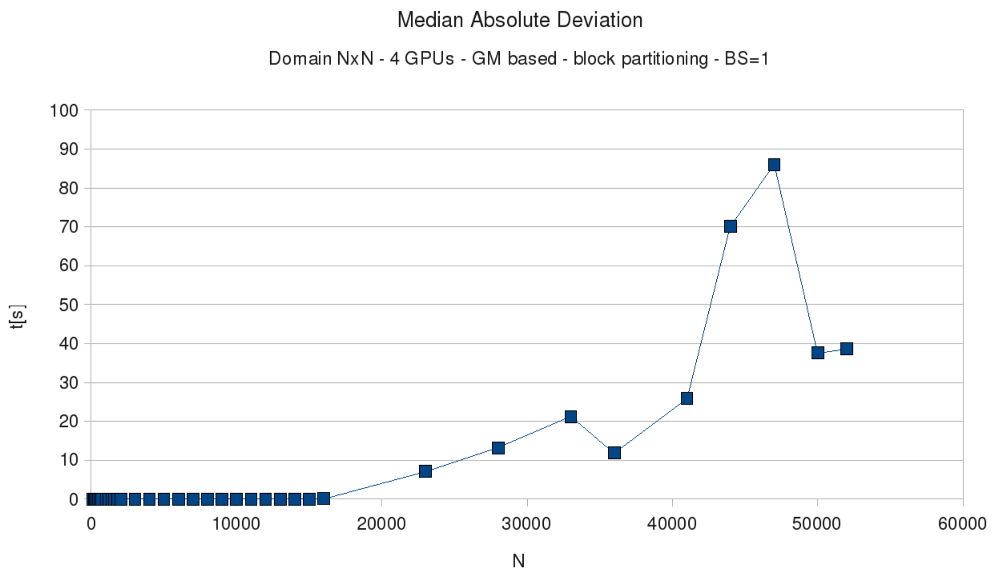

Median Elapsed Time
Domain NxN - 2 GPUs - GM based - BS=1
t[s]
N
MET
KT
ST
TT
CT
120
100
80
60
40
20
0
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000


Median Absolute Deviation
Domain NxN - 2 GPUs - GM based - BS=1
t[s]
N
4.00
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000

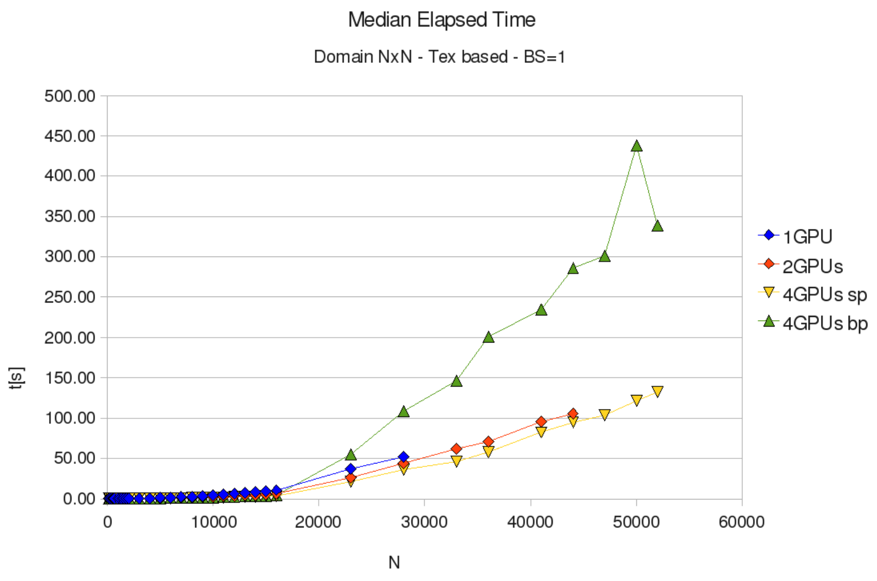

Median Elapsed Time
Domain NxN - 4 GPUs - GM based - strip partitioning - BS=1
t [s]
N
MET
KT
ST
TT
CT
0
20
40
60
80
100
120
140
10000
20000
30000
40000
50000
60000

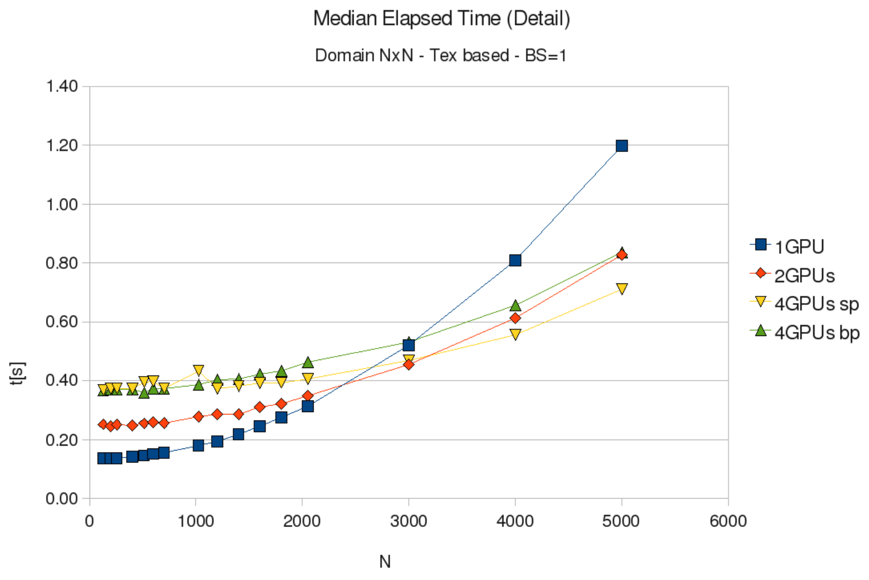

Median Absolute Deviation
Domain NxN - 4 GPUs - GM based - strip partitioning - BS=1
t[s]
N
0
1
2
3
4
5
6
7
10000
20000
30000
40000
50000
60000

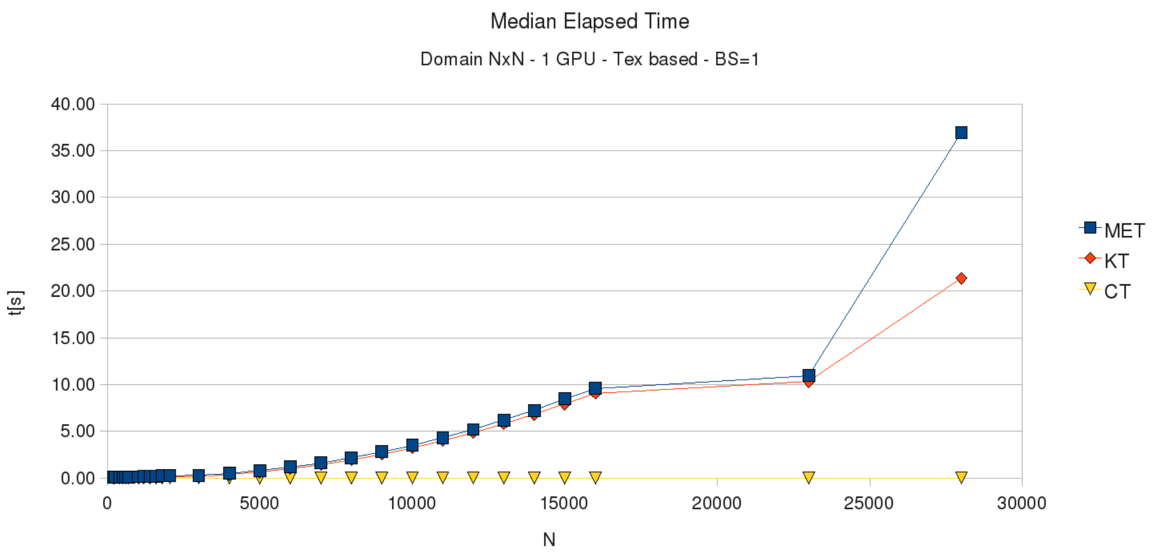

Median Elapsed Time
Domain NxN - 4 GPUs - GM based - block partitioning - BS=1
t[s]
N
MET
KT
ST
TT
CT
0
50
100
150
200
250
300
350
400
450
0
10000
20000
30000
40000
50000
60000

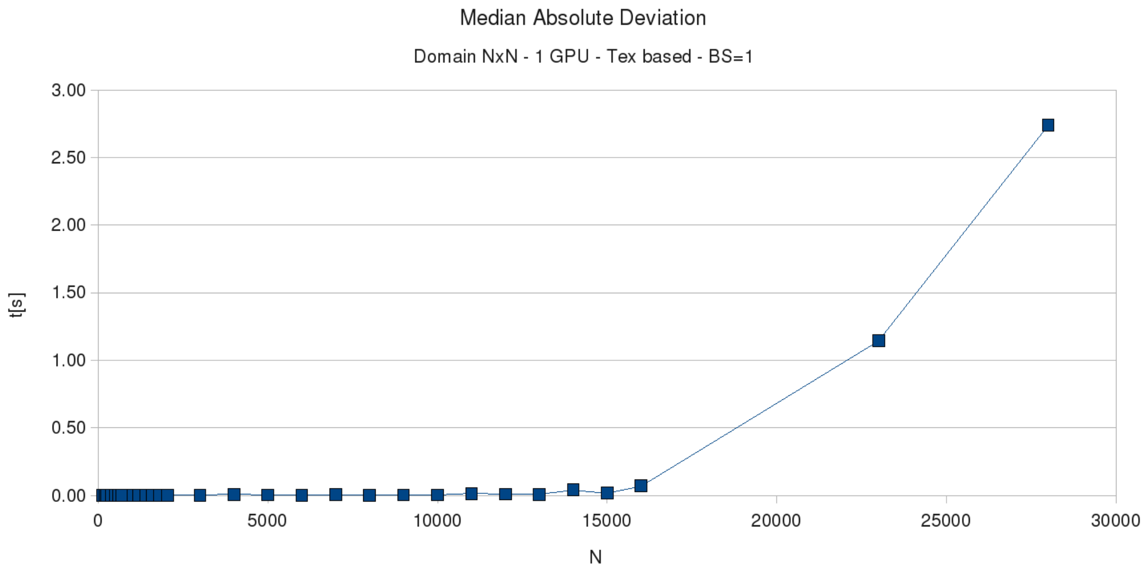

Median Absolute Deviation
Domain NxN - 4 GPUs - GM based - block partitioning - BS=1
t[s]
N
0
10
20
30
40
50
60
70
80
90
100
0
10000
20000
30000
40000
50000
60000

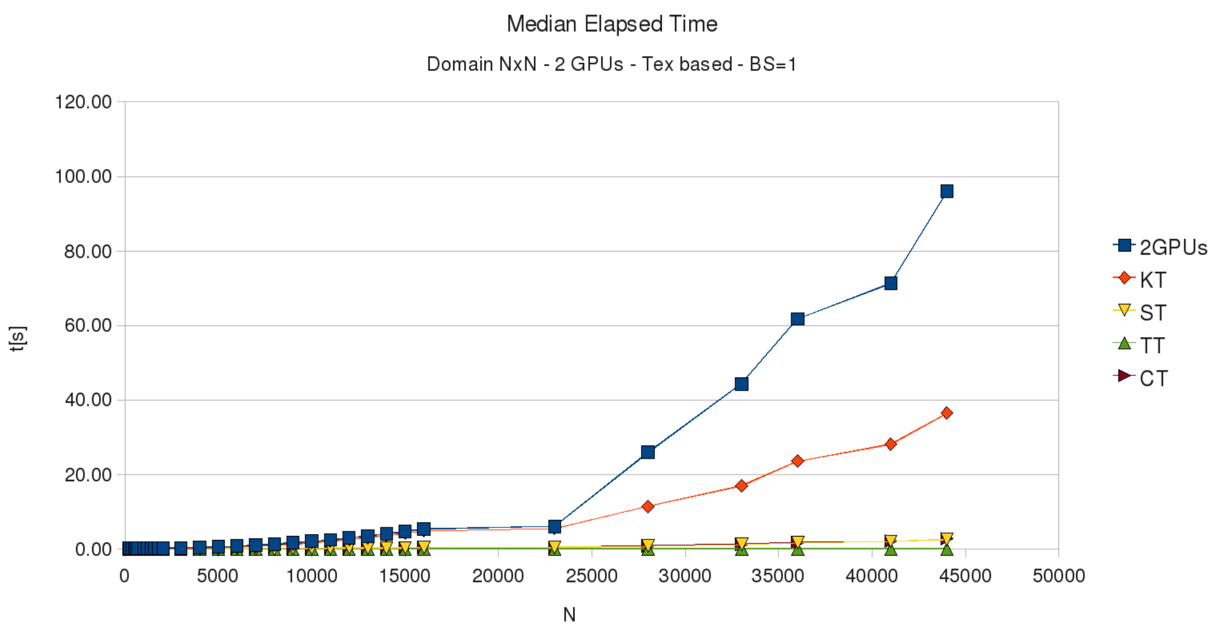
E.1.2 Texture-Based Kernels

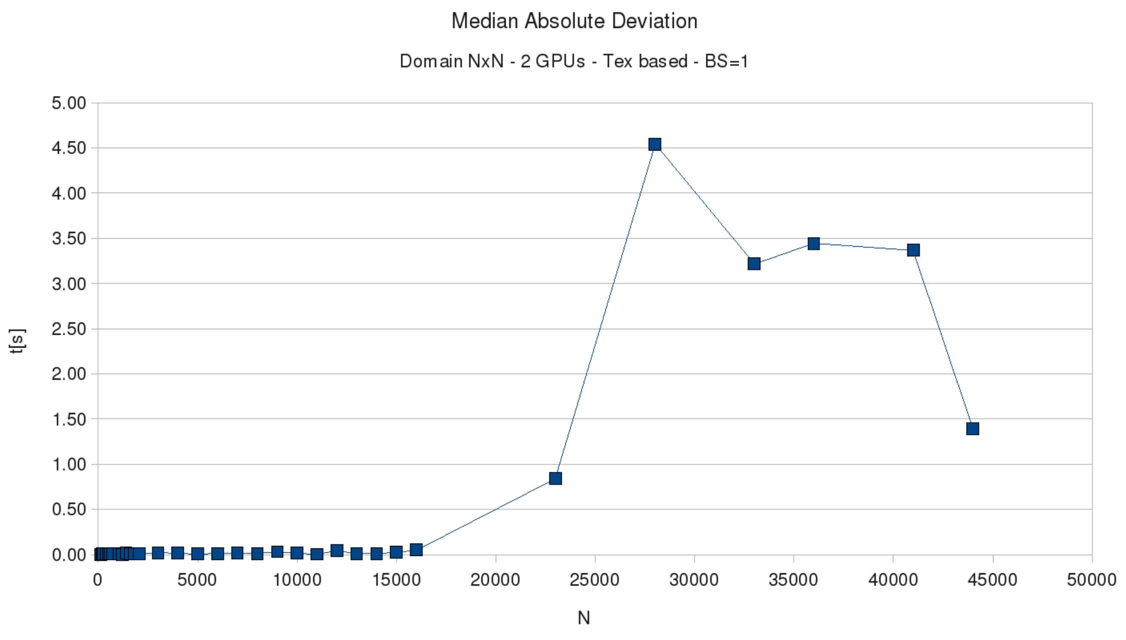


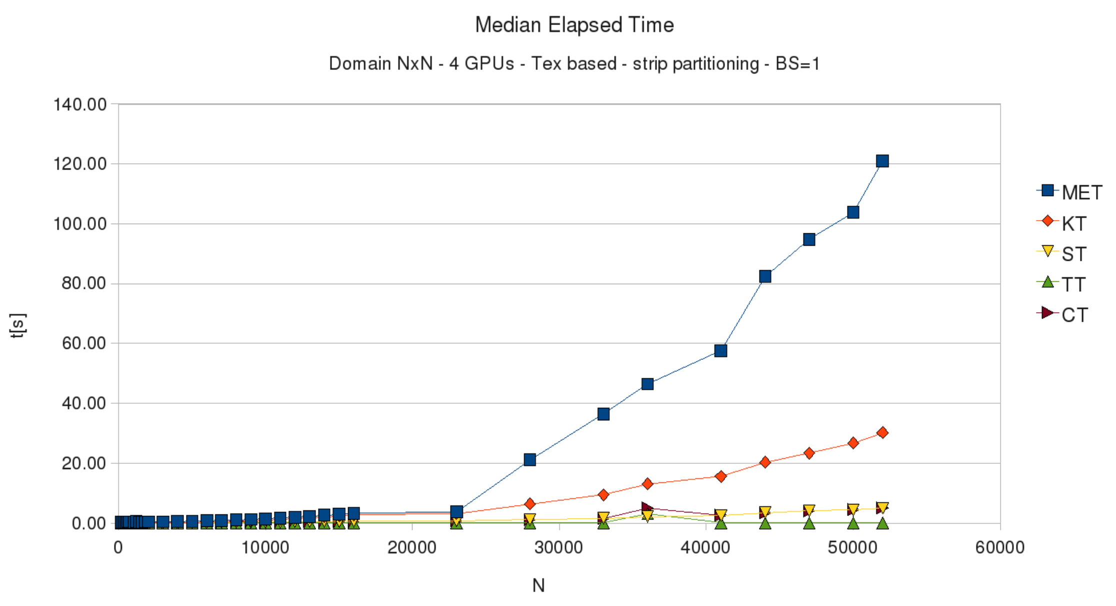


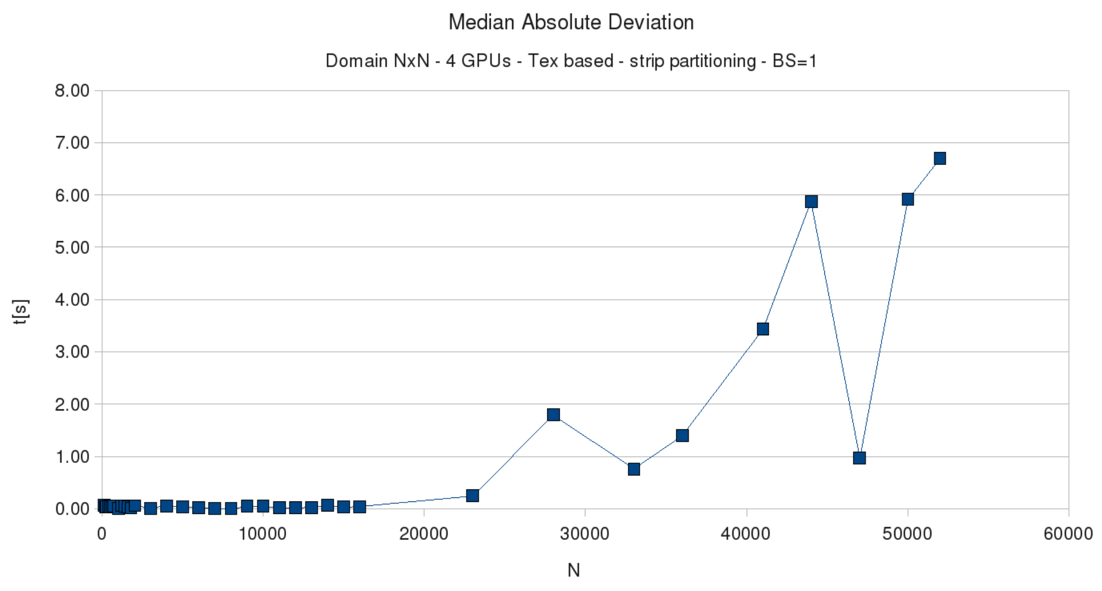

Median Elapsed Time
Domain NxN - 1 GPU - Tex based - BS=1
40.00
35.00
30.00
25.00
20.00
15.00
10.00
5.00
0.00
t[s]
0
5000
10000
15000
20000
25000
30000
N
MET
KT
CT

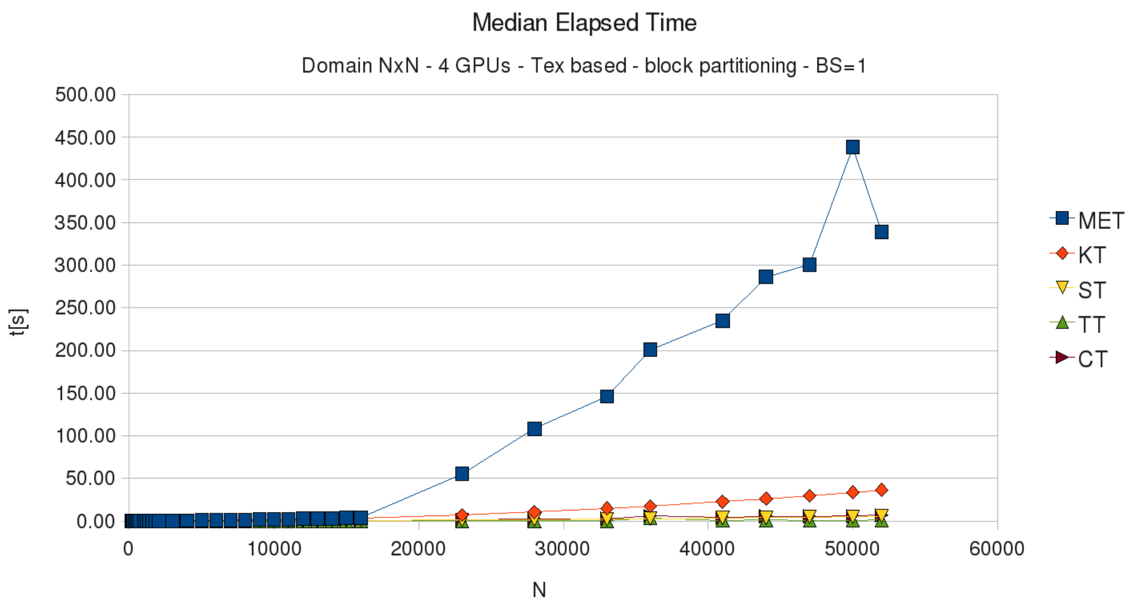

Median Absolute Deviation
Domain NxN - 1 GPU - Tex based - BS=1
3.00
2.50
2.00
1.50
1.00
0.50
0.00
t[s]
0
5000
10000
15000
20000
25000
30000
N

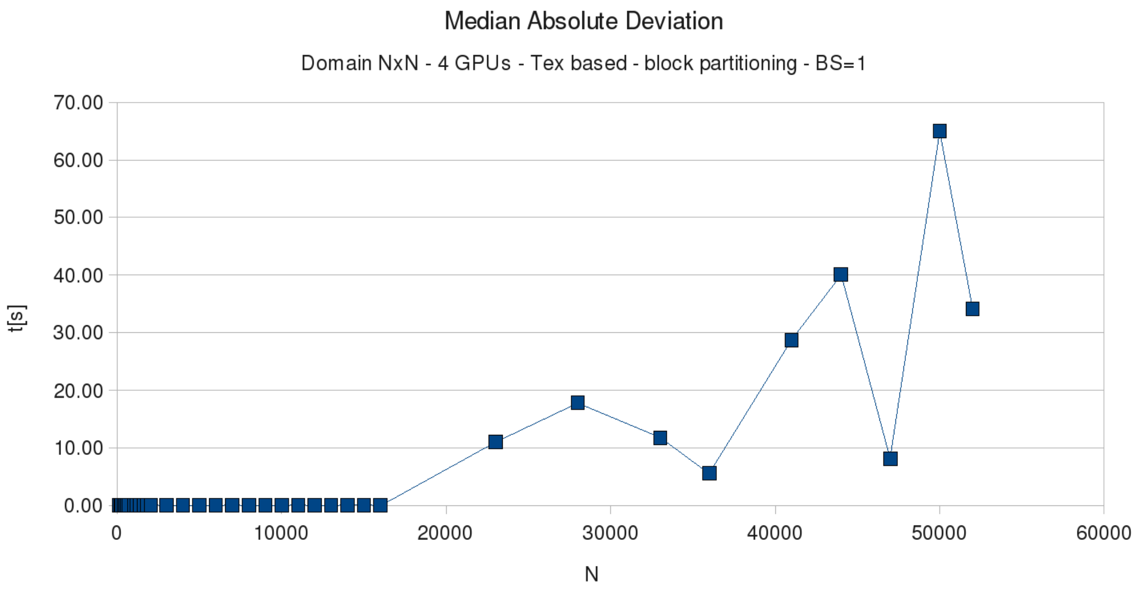

Median Elapsed Time
Domain NxN - 2 GPUs - Tex based - BS=1
t[s]
N
2GPUs
KT
ST
TT
CT
120.00
100.00
80.00
60.00
40.00
20.00
0.00
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000


Median Absolute Deviation
Domain NxN - 2 GPUs - Tex based - BS=1
t[s]
N
5.00
4.50
4.00
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000

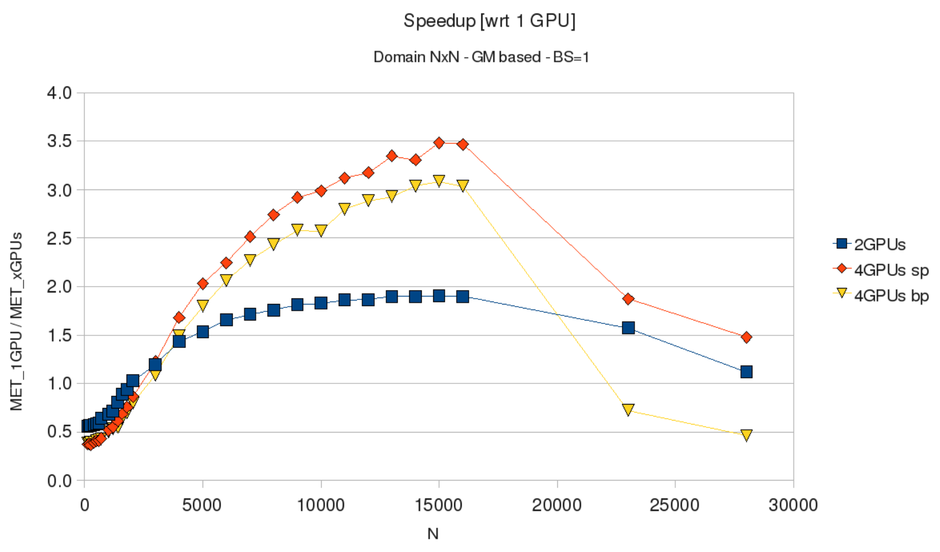

Median Elapsed Time
Domain NxN - 4 GPUs - Tex based - strip partitioning - BS=1
t[s]
140.00
120.00
100.00
80.00
60.00
40.00
20.00
0.00
0
10000
20000
30000
40000
50000
60000
N
MET
KT
ST
TT
CT

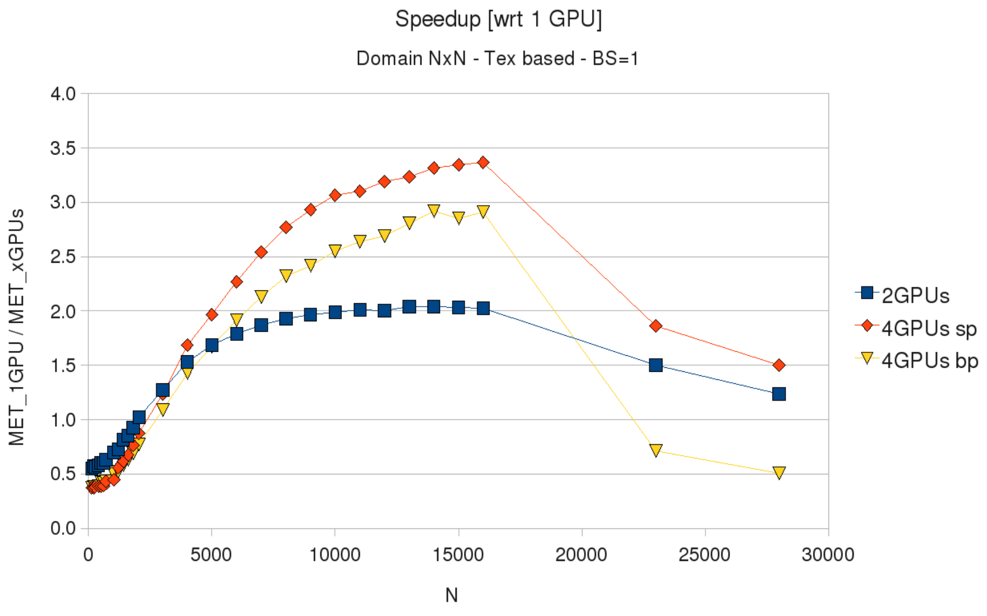

Median Absolute Deviation
Domain NxN - 4 GPUs - Tex based - strip partitioning - BS=1
t[s]
8.00
7.00
6.00
5.00
4.00
3.00
2.00
1.00
0.00
0
10000
20000
30000
40000
50000
60000
N

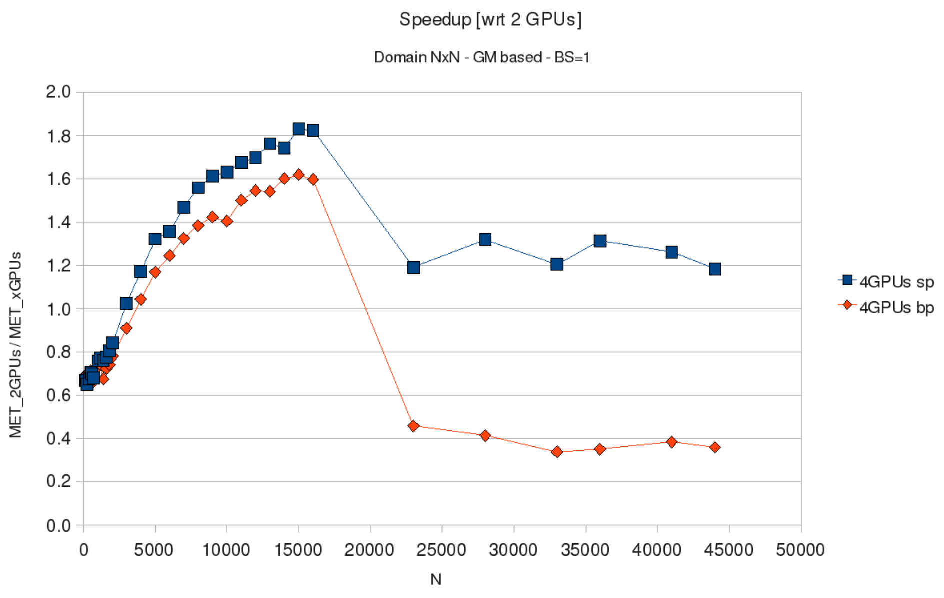

Median Elapsed Time
Domain NxN - 4 GPUs - Tex based - block partitioning - BS=1
500.00
450.00
400.00
350.00
300.00
250.00
200.00
150.00
100.00
50.00
0.00
t[s]
0
10000
20000
30000
40000
50000
60000
N
MET
KT
ST
TT
CT

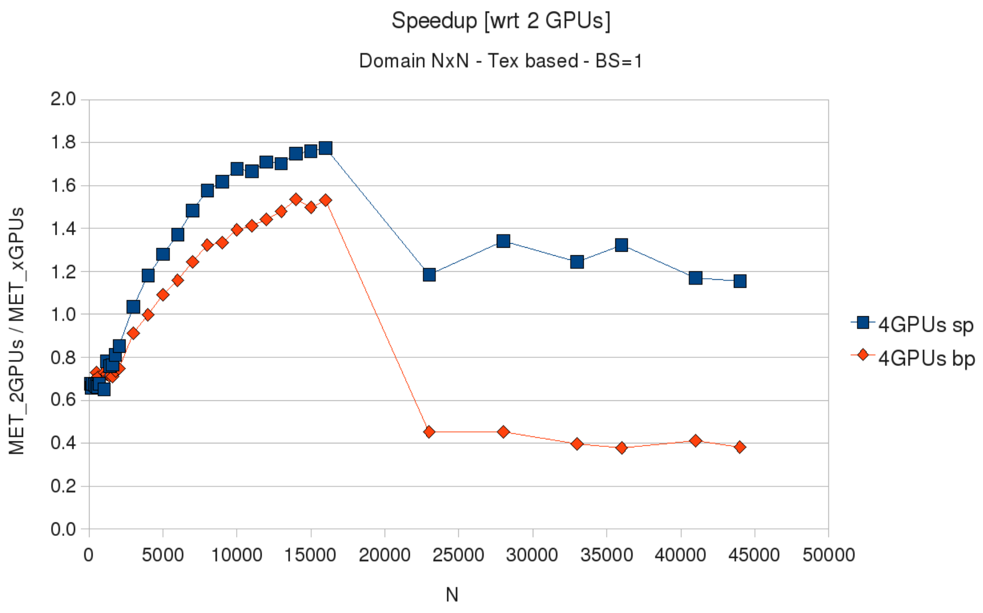

Median Absolute Deviation
Domain NxN - 4 GPUs - Tex based - block partitioning - BS=1
70.00
60.00
50.00
40.00
30.00
20.00
10.00
0.00
t[s]
0
10000
20000
30000
40000
50000
60000
N

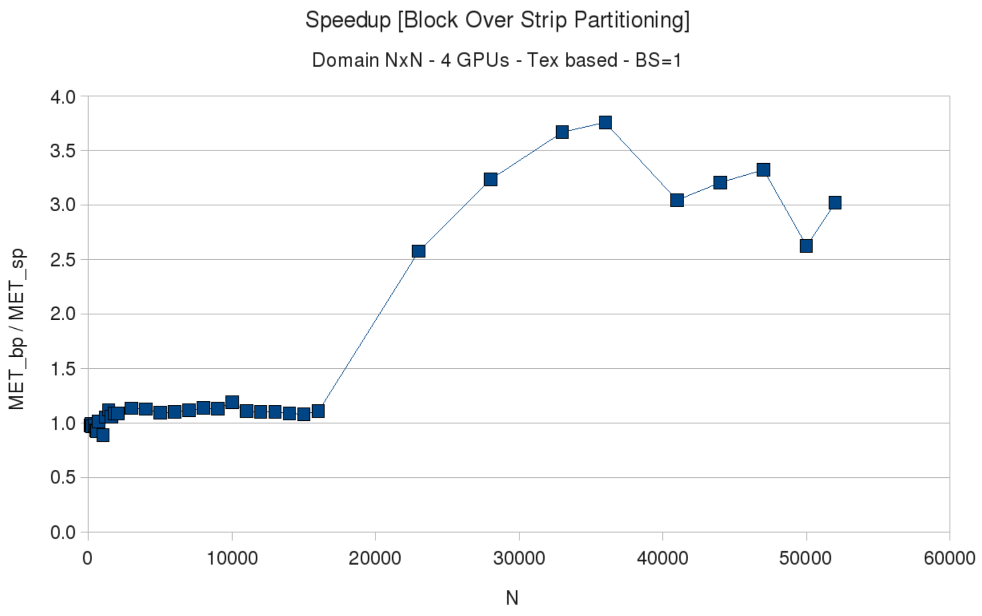
E.1.3 Speedups



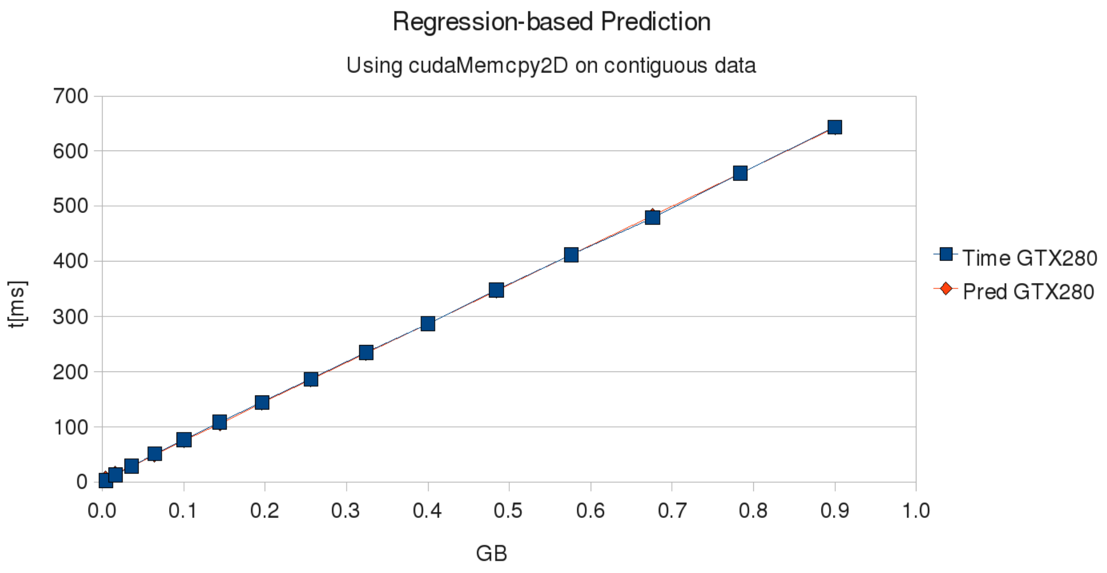


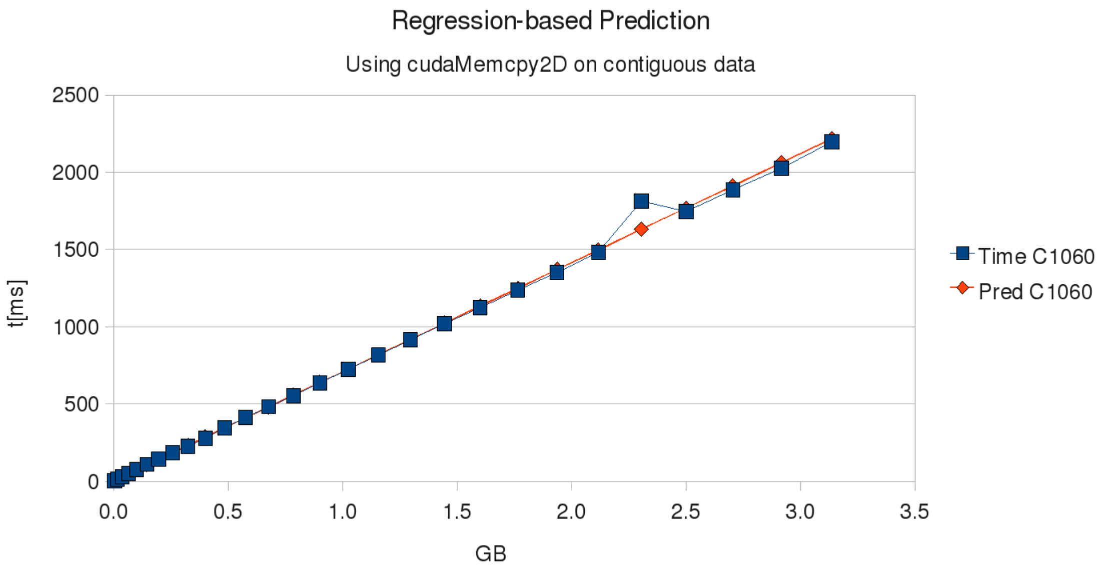

Speedup [wrt 2 GPUs]
Domain NxN - GM based - BS=1
MET_2GPUs / MET_xGPUs
N
4GPUs sp
4GPUs bp
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000


Speedup [wrt 2 GPUs]
Domain NxN - Tex based - BS=1
MET_2GPUs / MET_xGPUs
N
4GPUs sp
4GPUs bp
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000


Speedup [Block Over Strip Partitioning]
Domain NxN - 4 GPUs - Tex based - BS=1
MET_bp / MET_sp
N
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
10000
20000
30000
40000
50000
60000

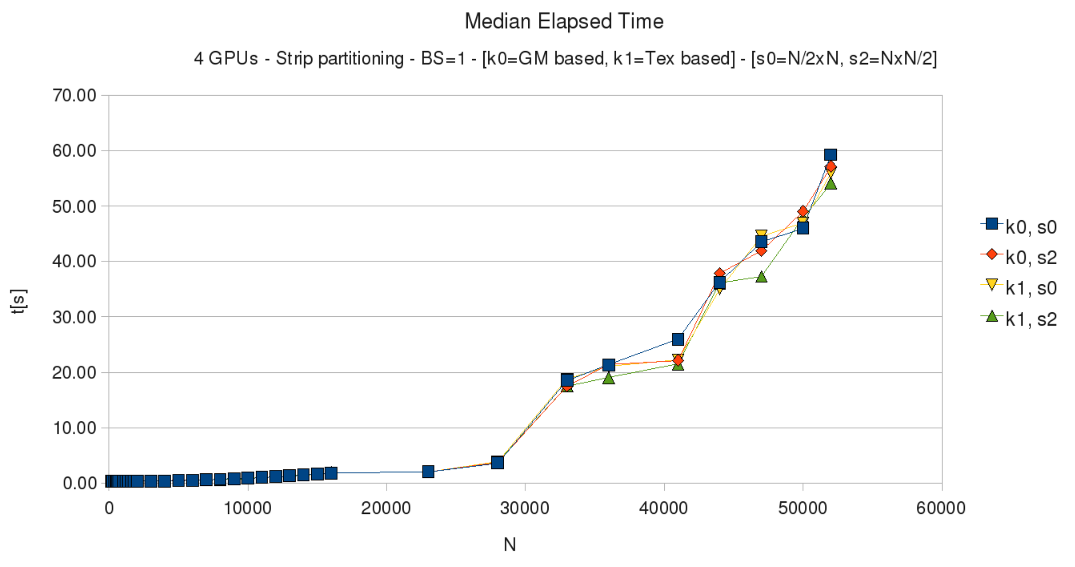
E.1.4 Regression-Based Predictions

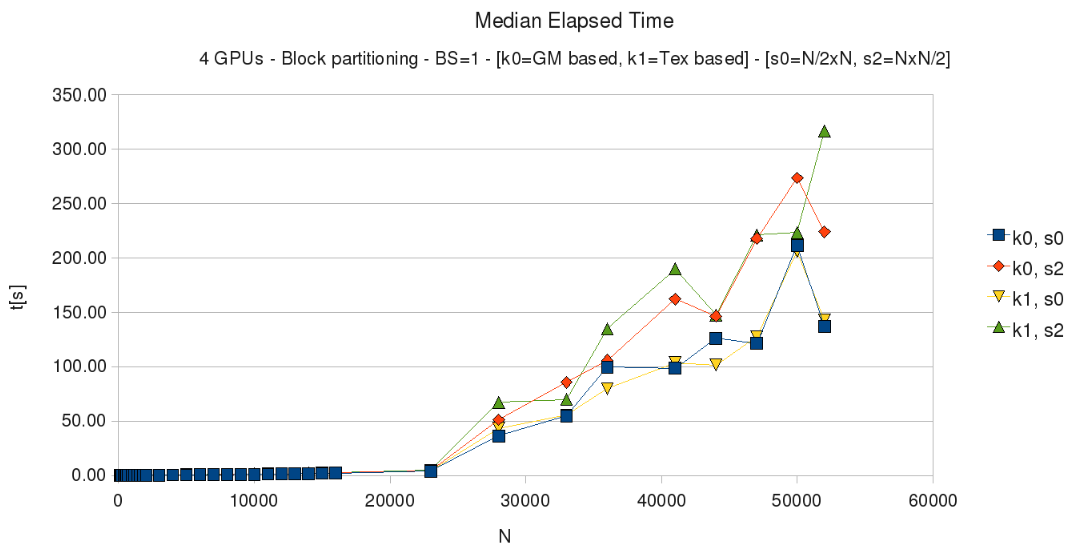


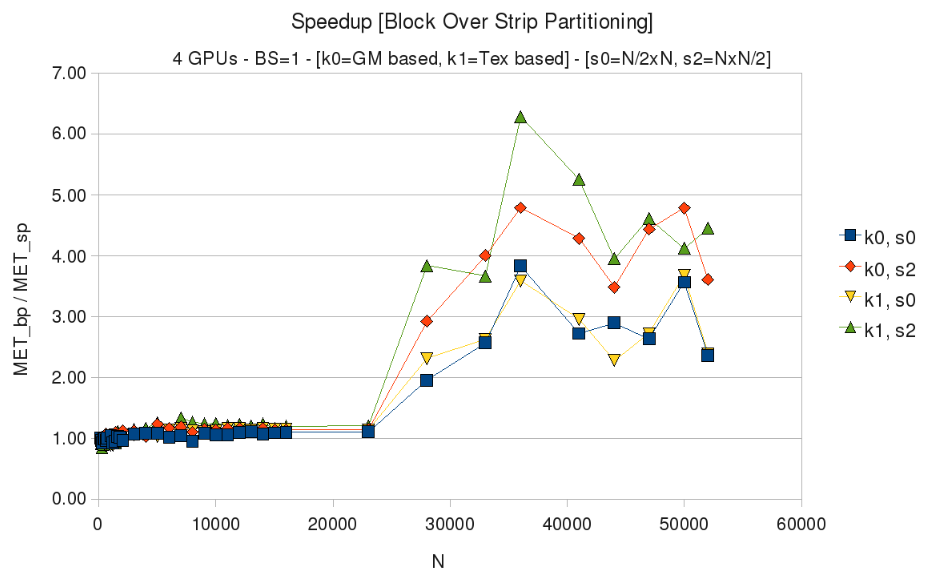


E.2 Execution Round 2

E.2.1 Execution Times and Speedups

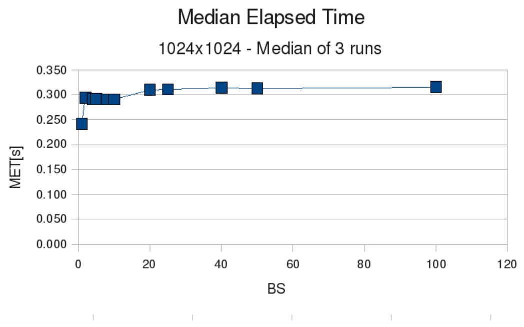


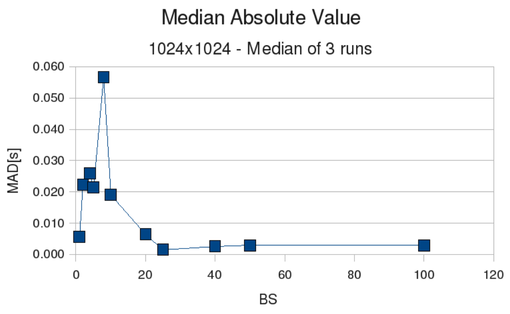


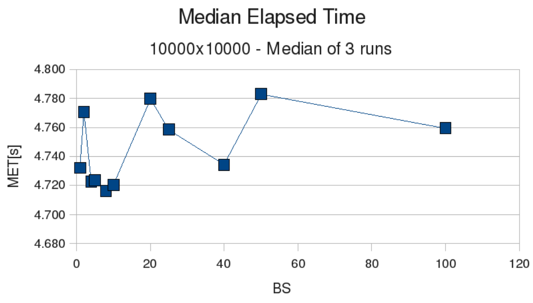

Speedup [Block Over Strip Partitioning]
4 GPUs - BS=1 - [k0=GM based, k1=Tex based] - [s0=N/2xN, s2=NxN/2]
MET_bp / MET_sp
0.00
1.00
2.00
3.00
4.00
5.00
6.00
7.00
0
10000
20000
30000
40000
50000
60000
N
k0, s0
k0, s2
k1, s0
k1, s2

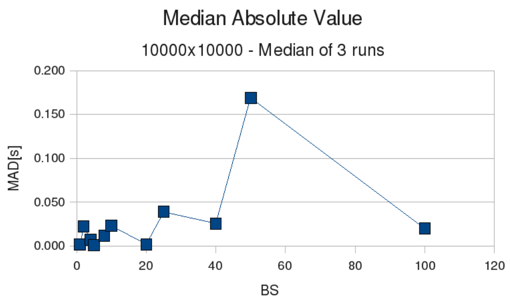
E.3 Execution Round 3

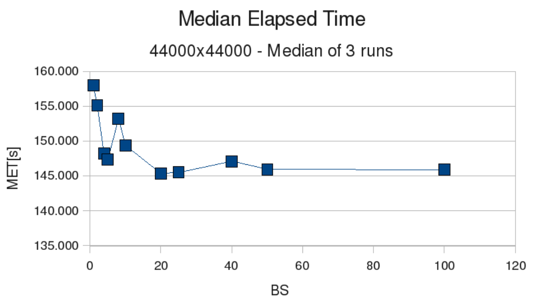
E.3.1 Execution Times

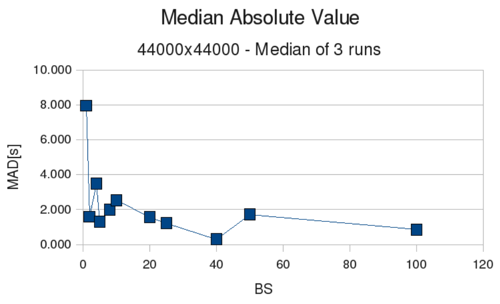












Appendix F

NOTUR Poster

NOTUR 2009 was the eighth edition of the annual meeting on High Performance Computing and Infrastructure for computational science in Norway. The meeting was intended for everyone that works with compute- and data-intensive applications. We had the opportunity to show a poster describing an earlier stage of our work.

The Conference was hold in Trondheim at the Norwegian University of Science and Technology (NTNU) in Trondheim on May 18-20, 2009.

Communication Challenges on Multi-GPU Systems

HPC Research Group

Daniele G. Spampinato
T.I.M.E. MS Student

Anne C. Elster
Advisor

Department of Computer and Information Science, NTNU

NTNU Norwegian University of Science and Technology

Introduction

Computing systems with multiple GPUs have a large computational resource at their disposal.
Theoretical peaks for recent commercial systems are in the order of several TFLOPS. However, even though such systems come with enormous computational power, applications' performance can still suffer from GPU-GPU and GPU-CPU communication overhead.

Case Study: Dirichlet Problem

Laplace equation $u_{xx} + u_{yy} = 0$

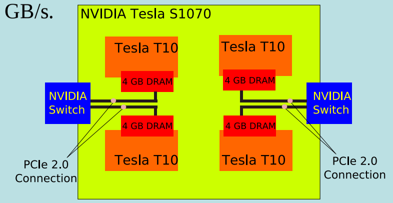
with Dirichlet boundary conditions:

$$u(x,y) \in W, \quad W \subset \mathbb{R}^2$$
$$u(x,y) = f(x,y), \quad \forall (x,y) \in \partial W$$

NVIDIA S1070 Computing System

- Four NVIDIA Tesla T10 GPUs.
- Each Tesla T10 has 240 processing cores.
- Theoretical peaks:
  - 4 TFLOPS single precision
  - 345 GFLOPS double precision

  $\}$ IEEE-754 compliant
- Each GPU connected to 4 GB DRAM 102 GB/s.
- Connection to the host via NVIDIA Switches and PCIe Host Interconnection Cards (HIC). Transfer rate up to 12.8 GB/s.

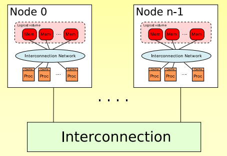


GOAL

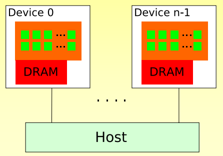
Evaluate domain decomposition methods that can improve performance of applications by harnessing the computational power of multiple GPUs simultaneously.

APPROACH

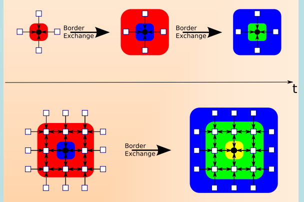
Communication model
Multi-GPU systems present analogous characteristics to SMP clusters.



$$\frac{GFLOPS}{MB/s} \approx k \cdot \frac{TFLOPS}{GB/s}$$

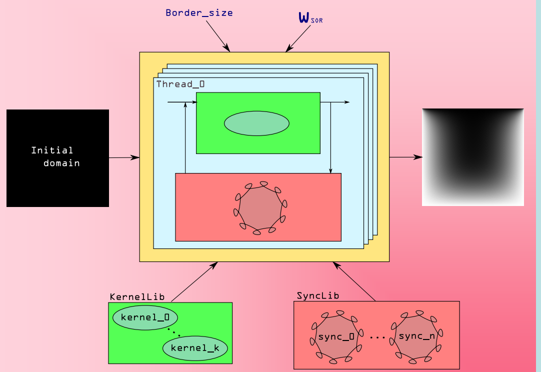


Extra calculations to reduce communication



5-points stencil red-black SOR

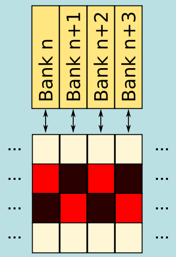
Benchmark application

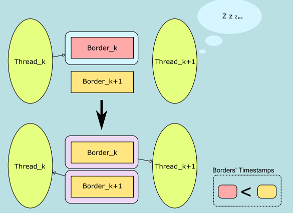


Preliminary Results

CUDA Kernel
Read from/write to GlobMem

Main objectives:
- Coalesced access to GlobMem
- Conflict-free access to ShMem
- 100% SM warp occupancy





Synchronization
GPUs handled by Pthreads.
Nonoptimized point-to-point synchronization via shared memory.

Up to 3Mpts/s with small border sizes using 2 GPUs

References

[1] R. Holtet, *Communications-reducing Stencil-based Algorithms and Methods*, Master's thesis, NTNU, Jul. 2003.
[2] D. M. Young, *Iterative Solution of Large Linear Systems*, Dover, 2003.

Acknowledgements

We thank NVIDIA for providing the graphics cards used in this project through Dr. Elster's membership in their Professor Affiliates Program.

Contacts

E-mail: daniele.spampinato@gmail.com
Web: http://www.idi.ntnu.no/~elster/hpc-lab/