2012

Parallelization & checkpointing of GPU applications through program transformation

Lizandro Damian Solano-Quinde
Iowa State University

Parallelization & checkpointing of GPU applications

through program transformation

by

Lizandro Damián Solano Quinde

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:

Arun Somani, Co-major Professor

Brett Bode, Co-major Professor

Akhilesh Tyagi

Zhao Zhang

Shashi Gadia

Iowa State University

Ames, Iowa

2012

DEDICATION

First and foremost I would like to dedicate this thesis to God, who gave me the strength to finish this part of my life. Also I want to dedicate this work to the most amazing and important human beings in my life, my family: my mom Gladys who taught me that important things in life are difficult to achieve, but with work and dedication everything is possible; my dad Galo who showed me that patience in life is the most valuable asset; my older brother Galito who showed me with his life to never give up even when things are so uncertain, and last but not least my younger brother Blady who, besides the things he taught me, has the extraordinary ability to draw a smile in my face everyday.

Thanks for your love, affection and constant support.

Lizandro Damián

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

This research would not be possible without the support of many people. First, I want to express my gratitude to my Major Professors Dr. Arun Somani and Dr. Brett Bode for their guidance, patience and insightful comments throughout this work. I also want to thank the people at the Scalable Computing Laboratory at the Ames Laboratory who made my life easier at work: Dr. Mark Gordon, Mark Klein, Nathan Humeston, Michelle Duncalf, Katie Walker and Dr. Alexander Gaenko. Special thanks to the best friends I could ever had: Verito Quintuña, Guimo Cabrera, Magali Mejía, Sonia Uguña, Pablo Astudillo, Raúl Pesántez and Hernán Quito, all of you made me realize that time and space do not end friendships. Finally, I want to thank my friends who made my life easier in Ames: Ramón Mercado, Claudia Navarro, Patricio Quiroga, Hakan Topakkaya, Patricio Galdames, Sai Talamudupula, Hugo Villegas, Garold Martin, and all my ecuadorian friends.

This work was performed at the Ames Laboratory under contract number DE-AC02-07CH11358 with the U.S. Department of Energy. The document number assigned to this thesis/dissertation is IS-T 3081.

ABSTRACT

GPUs have emerged as a powerful tool for accelerating general-purpose applications. The availability of programming languages that makes writing general-purpose applications for running on GPUs tractable have consolidated GPUs as an alternative for accelerating general-purpose applications. Among the areas that have benefited from GPU acceleration are: signal and image processing, computational fluid dynamics, quantum chemistry, and, in general, the High Performance Computing (HPC) Industry.

In order to continue to exploit higher levels of parallelism with GPUs, multi-GPU systems are gaining popularity. In this context, single-GPU applications are parallelized for running in multi-GPU systems. Furthermore, multi-GPU systems help to solve the GPU memory limitation for applications with large application memory footprint. Parallelizing single-GPU applications has been approached by libraries that distribute the workload at runtime, however, they impose execution overhead and are not portable. On the other hand, on traditional CPU systems, parallelization has been approached through application transformation at pre-compile time, which enhances the application to distribute the workload at application level and does not have the issues of library-based approaches. Hence, a parallelization scheme for GPU systems based on application transformation is needed.

Like any computing engine of today, reliability is also a concern in GPUs. GPUs are vulnerable to transient and permanent failures. Current checkpoint/restart techniques are not suitable for systems with GPUs. Checkpointing for GPU systems present new and interesting challenges, primarily due to the natural differences imposed by the hardware design, the memory subsystem architecture, the massive number of threads, and the limited amount of synchronization among threads. Therefore, a checkpoint/restart technique suitable for GPU systems is needed.

The goal of this work is to exploit higher levels of parallelism and to develop support for

application-level fault tolerance in applications using multiple GPUs. Our techniques reduce the burden of enhancing single-GPU applications to support these features. To achieve our goal, this work designs and implements a framework for enhancing a single-GPU OpenCL application through application transformation.

CHAPTER 1. Introduction

1.1 Motivation

The ever increasing demand for computational power imposed by computer applications has motivated the design of faster microprocessors. Processor performance is achieved primarily by exploiting two factors:

- *Instruction Level Parallelism (ILP)*, achieved by pipelining the operations on the processor, or by multiple instruction issue. The limits on ILP are given by data and control hazards and dependencies [1].
- *Memory Latency*, reduced by minimizing the number of data transfers from memory, which is achieved by adding several levels of on-chip cache memory. Cache memory exploits temporal and spatial data locality, thus, cache memories are less effective if a program does not present temporal and spatial data locality properties [1, 2].

To increase the ILP and reduce memory latency, processor designers have used several techniques, such as: out-of-order execution, branch prediction, speculative execution, multiple instruction issue, pipelining, data and instruction caching, and data prefetching, among other techniques. These techniques require additional control logic, which results on the need of more transistors integrated on a single chip. Technology scaling allows for integrating in a single chip twice the number of transistors approximately every two years [3, 4].

Under this scenario, the software industry has achieved better performance on its software applications at every new processor generation without any major change on the computation model, but only by taking advantage of faster processors.

Limits on Performance. During the last few years, processor performance has been limited by the so-called *"Brick Wall for Serial Performance"*, defined by David Patterson as [5]:

Brick Wall for Serial Performance = ILP Wall+Memory Wall+Power Wall

- *The ILP wall*, refers to the fact that it is increasingly more difficult to find enough parallelism on the instructions. Besides, ILP increases hardware complexity which raises power consumption.
- *The Memory wall*, processor clock rates have been increasing faster than memory clock rates, i.e. memory latency is on the order of hundred of clock cycles.
- *The Power wall*, doubling every two years the number of integrated transistors on a single chip increases both the dynamic power and the static power. Power is becoming a concern not only because it is reaching high levels, but also because of the practical ability for dissipation of the the generated heat.

In 2002, Andrew Grove, chairman at the board of Intel, stated at the Electron Devices Meeting that the brick wall, particularly the power brick, is "becoming a limiter of integration" [6]. Some projections made by Intel forecast that by increasing the transistor integration, processors would only obtain a modest amount of performance improvement compared to the power increase. For instance: a 13% performance improvement increases aproximately 70% the power; on the other hand by decreasing 13% performance, a decrease of roughly 50% of power is achieved [7].

This observation implies that instead of having only one single high frequency processor, or core, it is more profitable, in terms of power and computing performance, integrating several lower frequency cores in the same chip, i.e *Chip Multicore Processors (CMPs)*.

Chip Multicore & Manycore Processors. Through multiple low frequency cores CMPs achieve higher power efficiency, while still executing more instructions per cycle than unicore processors. Chip manufacturers have successfully integrated two, four and eight cores

into a single chip for comercial CMPs [8], and the trend is to continue to integrate hundred of cores in a single chip, i.e *Manycore Processors* [9]. Manycore processors are not only present as the main processor of the computer system, but also they are present as the so-called Hardware Accelerators (HWAs).

HWAs have been designed and deployed to release the main processor of time-consuming tasks, mainly floating point operations and graphics rendering, leaving the main processor free to execute other tasks. Currently, graphics and FPGA-based accelerators are most common.

FPGA[1]-based accelerators have become popular because of their reconfigurability, performance and suitability for working in parallel. Potentially FPGA-based accelerators contain tens of accelerator units working in parallel.

Graphics accelerators emerged around 1984 providing basic fixed 2D primitives, and evolved to provide full programmable video capabilities. In 1999, Graphics accelerators turned into manycore processors, known as Graphics processing Units (GPUs) [10]. Current GPUs integrate hundred of processing cores and achieve up to 1.3 TFlops for single precision and over 500 GFlops for double precision. In addition to this tremendous computational power, GPUs have become fully programmable, which makes GPUs amenable for accelerating massively-parallel general purpose applications, opening a new application field to GPUs called *General-Purpose Computation on GPUs* (GPGPU).

General-Purpose Computation on GPUs. The availability of programming languages that makes writing general-purpose applications for running on GPUs tractable have consolidated GPUs as an alternative for accelerating general-purpose applications. CUDA (Computing Unified Device Architecture) developed by Nvidia targets general purpose programming on its GPUs. OpenCL (Open Computing Language) developed by the Khronos group is a non-proprietary programming language targeting any parallel platform.

In spite of the availability of GPGPU-oriented programming languages, exploiting the huge computational capabilities of the GPU requires careful algorithm redesign and application rewrite. Evidently not all applications are suitable for GPU acceleration, only applications that

[1]FPGA (Field Programmable Gate Array) is a set of logic gates programmed to fulfill user-specific tasks

are parallel in nature are suitable for execution on a GPU. Currently, applications modified and optimized to run on GPU-accelerated systems range over a variety of fields, such as: signal and image processing, computational fluid dynamics, quantum chemistry, and, in general, the High Performance Computing (HPC) Industry [11].

Multi-GPU Systems. The computer industry is developing systems with GPU accelerators specially optimized for HPC (High Performance Computing) applications, integrating one, two or four GPUs to enhance performance. The NVidia Tesla K10 is a HPC-optimized computing system that integrates two Tesla GK104 GPUs and provides a theoretical peak performance of 4.57 TFlops for single precision and 190 GFlops for double precision.

The natural next step towards exploiting higher levels of parallelism using GPUs is to parallelize single-GPU applications for running in multi-GPU systems. Furthermore, multi-GPU systems help to solve the GPU memory limitation for applications with large application memory footprint. Multi-GPU execution can be achieved by exploiting task and/or data parallelism, the former is easy to achieve but it might lead to poor load balance, while the later achieves better load balance but it requires data and computation decomposition, which adds complexity.

The potential for achieving higher levels of parallelism or overcoming the GPU memory limitation added to the complexity of re-writing or modifying applications for supporting execution on multi-GPU systems encourages the research of a framework that minimize the manual transformation of applications.

Reliability of GPU Systems. Like any computing device, GPUs are not free of transient and permanent failures. A study conducted on more than 20,000 GPUs on the Folding@home distributed computing network shows that two-thirds of tested GPUs exhibit susceptibility to transient errors on memory or logic. This study suggests the utilization of hardware or software fault tolerant techniques to mitigate the impact of failures in GPUs [12]. Although no hard data on permanent errors for GPUs has been published, GPUs are exposed to the same issues as unicore processors and memories.

Application execution time is affected negatively by the presence of failures, because all the work done up to the failure time is lost, and, consequently, the application should be restarted. Fault tolerance schemes prevent lost work by saving application information before the failure occurs.

Checkpoint/Restart for GPU Systems. The most popular fault tolerant scheme is the Checkpoint/Restart technique [13]. This scheme takes periodic snapshots of the application state (checkpoints) during its execution. After an application crashes, the checkpoints taken before the application crash are utilized to restart the application from the last valid checkpoint.

Checkpointing for unicore and multi-processors have been largely studied over the past two decades [13, 14, 15, 16, 17]. However, checkpointing for GPU systems presents different issues compared to those issues arisen by unicore and multi-processor systems. These differences emerge from the natural differences between the hardware design and programming languages of GPUs compared to those of unicore processors and multi-processors, as illustrated next.

- GPUs have separate memory and the internal memory organization is different than memory on CMPs or multi-processors,
- GPUs require a massive number of threads to achieve peak performance, and
- GPUs provide limited synchronization.

These differences make current checkpoint schemes unsuitable for GPU systems. The growing popularity of GPU systems in the scientific field, encourages the research of fault tolerant schemes suitable for GPU systems.

1.2 Problem Statement

GPUs have emerged as a powerful tool for accelerating general-purpose applications. The availability of programming languages that makes writing general-purpose applications for running on GPUs tractable have consolidated GPUs as an alternative for accelerating general-purpose applications. Among the areas that have benefited from GPU acceleration are: signal

and image processing, computational fluid dynamics, quantum chemistry, and, in general, the High Performance Computing (HPC) Industry.

In order to continue to exploit higher levels of parallelism with GPUs, multi-GPU systems are gaining popularity. In this context, single-GPU applications are parallelized for running in multi-GPU systems. Furthermore, multi-GPU systems help to solve the GPU memory limitation for applications with large application memory footprint. Parallelizing single-GPU applications has been approached by libraries that distribute the workload at runtime, however, they impose execution overhead and are not portable. On the other hand, on traditional CPU systems, parallelization has been approached through application transformation at pre-compile time, which enhances the application to distribute the workload at application level and does not have the issues of library-based approaches. Hence, a parallelization scheme for GPU systems based on application transformation is needed.

Like any computing engine of today, reliability is also a concern in GPUs. GPUs are particularly vulnerable to transient and permanent failures. Current checkpoint/restart techniques are not suitable for systems with GPUs. Checkpointing for GPU systems present new and interesting challenges, primarily due to the natural differences imposed by the hardware design, the memory subsystem architecture, the massive number of threads, and the limited amount of synchronization among threads. Therefore, a checkpoint/restart technique suitable for GPU systems is needed.

The issues of multi-GPU execution and application-level checkpointing for single-GPU applications are covered on this work.

1.3 Goal and Objectives

1.3.1 Goal

The goal of this work is to exploit higher levels of parallelism and to develop support for application-level fault tolerance in applications using multiple GPUs. Our techniques reduce the burden of enhancing single-GPU applications to support these features. Multi-GPU execution not only exploits higher levels of parallelism but also helps to overcome the global

memory limitation. Fault tolerance, i.e. application-level checkpointing, enables GPU systems to reliably run real long-time scientific applications.

1.3.2 Objectives

To achieve our goal, this work designs and implements a framework for enhancing a single-GPU OpenCL application through application transformation. The enhancements supported are: (i) multi-GPU execution and (ii) application-level checkpointing. In this context, the objectives of this research can be divided in four major areas: *acceleration in single-GPU and multi-GPU systems*, *computation and data decomposition*, *minimizing checkpoint overhead* and *automated application transformation*.

1.3.2.1 Acceleration in Single-GPU and Multi-GPU Systems

- Achieving high performance in GPU systems relies on high occupancy and efficient global memory access. This work analyzes the effect of these parameters on the algorithm for unstructured grid-based analysis in terms of memory access pattern and GPU occupancy. Based on this analysis an optimized algorithm suitable for running on GPUs is proposed.

- The performance enhancement achieved by multi-GPU systems depends on the suitability of the application algorithm to exploit data or task parallelism and keep communication overhead low. This research analyzes the data and task parallelism techniques, applied to the unstructured grid application, in terms of the communication overhead and data dependencies, aiming to determine which parallelism technique achieves better results.

1.3.2.2 Computation and Data Decomposition

- Computation and data decomposition are the building blocks for implementing data parallelism and application-level checkpointing. In this context, computation and data decomposition depend on the application data flow, i.e. data dependencies, and the access patterns. In particular, these two characteristics (i) limit the parallelism achieved and impose data exchange among GPUs, i.e. communication overhead, and (ii) limit the

ability of hiding checkpoint latency. This research aims to study computation and data decomposition in terms of data dependencies and data access patterns, and to propose mechanisms to automate this decomposition.

- The communication overhead imposed by the decomposition, is aggravated by the performance of the network that interconnects the GPU devices. This research investigates the effect on the network performance, in terms of the bandwidth degradation, as well as the benefit of overlapping computation and communication to reduce the overall overhead achieved.

1.3.2.3 Minimizing the Checkpoint Overhead

- In an iterative application, minimizing the checkpoint overhead per iteration is achieved by finding a checkpoint location where the checkpoint latency can be reduced by partitioning data and computation and overlapping checkpoint operations with computation. To this end, this research aims to provide metrics for selecting a checkpoint location where the checkpoint overhead is minimized through overlapping computation and checkpoint operations.

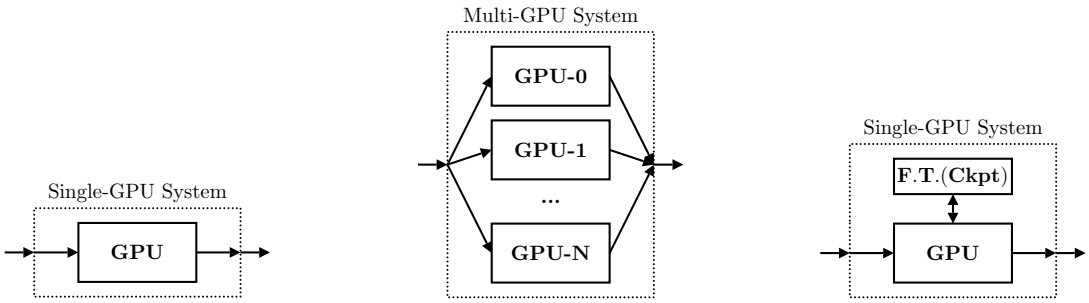
- Minimizing the total checkpoint overhead is achieved by finding a checkpoint period that keeps a balance among the total overhead and the amount of information lost. This research proposes a checkpoint interval based on the probability of failure.

1.3.2.4 Automated Application Transformation

- Depending on the feature desired (multi-GPU execution or application-level checkpointing) the transformation requirements are different. To reduce the burden of enhancing GPU applications, this research aims for identifying and automating the transformation needed by the application to support the desired feature.

1.4 Contributions

We develop a framework to enhance single-GPU applications. The enhancements supported are: multi-GPU execution and fault tolerance (application-level checkpointing), see Figure 1.1.



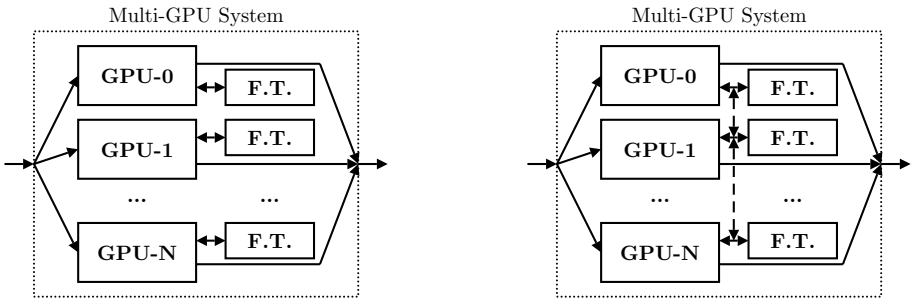
(a) Single-GPU Application (b) Multi-GPU Execution (c) Fault Tolerance

Figure 1.1: Enhancements performed to a single-GPU application

We make the following contributions:

1. **Exploiting Higher Levels of Parallelism**
   - Applications can exploit higher levels of parallelism, potentially allowing higher speedups.
   - The application memory footprint is divided between multiple GPUs, potentially overcoming the memory limitation imposed by the size of the GPU global memory.
2. **Increasing Fault Tolerance**
   - Application-level checkpointing enables GPU systems to reliably run real long-time scientific applications.
   - Kernel and data decomposition as well as checkpoint operations overlapping techniques reduce the checkpoint overhead.
3. **Automated Application Transformation**
   - Reduces the effort of enhancing single-GPU application for supporting multi-GPU execution and application-level checkpointing.

As it can be seen in Figure 1.2, our checkpointing scheme can be applied to multi-GPU systems, however, at the current time no coordination among the checkpoints taken is implemented. Coordination among the checkpoints taken will save a consistent state of the application and might reduce the total checkpoint overhead by avoiding checkpointing duplicate information.



(a) Non Coordinated (b) Coordinated

Figure 1.2: Fault tolerance applied to a multi-GPU application

1.5 Overview of Dissertation

This chapter starts by presenting the performance issues of single core processors that motivated the development of multi-core and many-core processors, i.e. CMPs and GPUs. GPUs gained popularity to accelerate general-purpose applications, however, issues related to reliability and multi-GPU execution remain open. This research addresses these issues and proposes a framework for enhancing single-GPU application to support application-level checkpointing and multi-GPU execution. The rest of this document is organized as follows.

Chapter 2 presents current approaches for multi-GPU execution and fault tolerance, as well as a background on GPU architecture and failures. Chapter 3 introduces the algorithm of unstructured grid-based applications and its implementation on single-GPU systems, the implementation on multi-GPU systems is presented in Chapter 4. Chapter 5 introduces the data analyzer of our framework for program transformation. Subsequently, Chapter 6 presents the framework for program transformation to parallelize single-GPU applications. Chapter 7 introduces the main techniques for application-level checkpointing. Following this, Chapter 8

presents the framework for program transformation to support application-level checkpointing of single-GPU applications. Finally, Chapter 9 covers the concluding remarks and future work.

CHAPTER 2. Related Work

2.1 Introduction

The release of powerful GPUs with a fully programmable pipeline attracted the interest of the research community for accelerating general-purpose applications using GPUs. The Nvidia Fermi integrates 512 cores and achieves a single precision peak performance of 1.3 TFlop.

Moreover, the availability of programming languages, such as CUDA and OpenCL, that makes writing applications for GPUs tractable have consolidated GPUs as an alternative for accelerating general-purpose applications, opening a new field for GPUs called General-Purpose Computing on GPUs. Areas that have benefited of GPU acceleration includes: linear algebra, molecular dynamics, computational fluid dynamics, quantum chemistry, signal processing, among others.

Multi-GPU systems is the natural next step for achieving higher levels of parallelism. Furthermore, multi-GPU systems help to overcome the global memory limitation for executing applications with large application memory footprint. The achievable speedup is limited by the communication overhead and data dependencies.

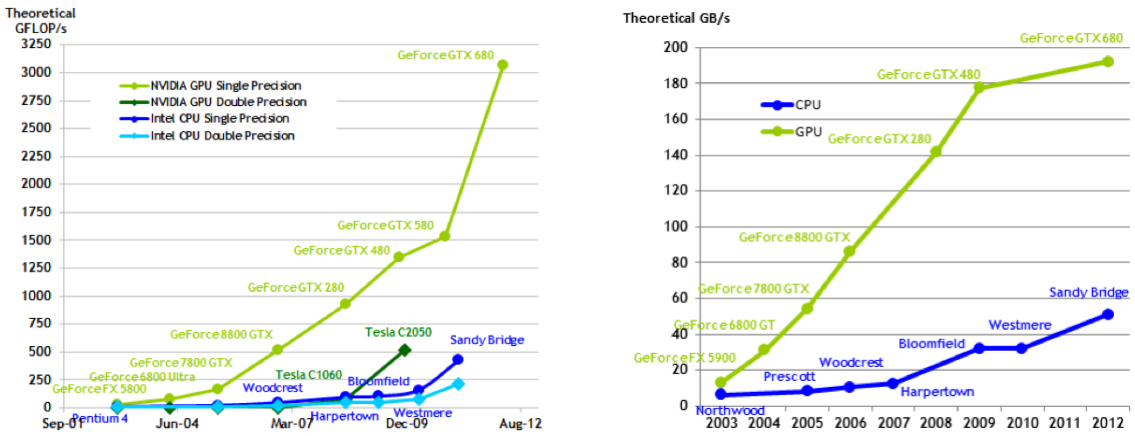
As with any other computational device, reliability in GPUs is also a concern because of the occurrence of soft and hard failures. A study conducted on more than 20,000 GPUs on the Folding@home distributed computing network shows that two-thirds of the tested GPUs exhibit susceptibility to transient errors on memory or logic [12]. In this context, the checkpoint/restart technique is commonly utilized for providing fault tolerance.

2.2 GPU Architecture

During the last few years the game industry has evolved such that current games present great complexity and realistic graphics. Such complex and realistic graphics require hardware with huge computation capabilities and bandwidth, which is provided by Graphics Processing Units (GPUs).

2.2.1 CPUs vs GPUs

A GPU is a massively parallel, multi-threaded, manycore processor with tremenduous computational capabilities and high memory bandwidth. Figure 2.1 shows a comparison between GPUs and CPUs[1] computation capabilities and bandwidth, where it can be seen that, as of 2011, the peak performance and bandwidth of the GPU is about four times the performance and bandwidth of an Intel CPU.



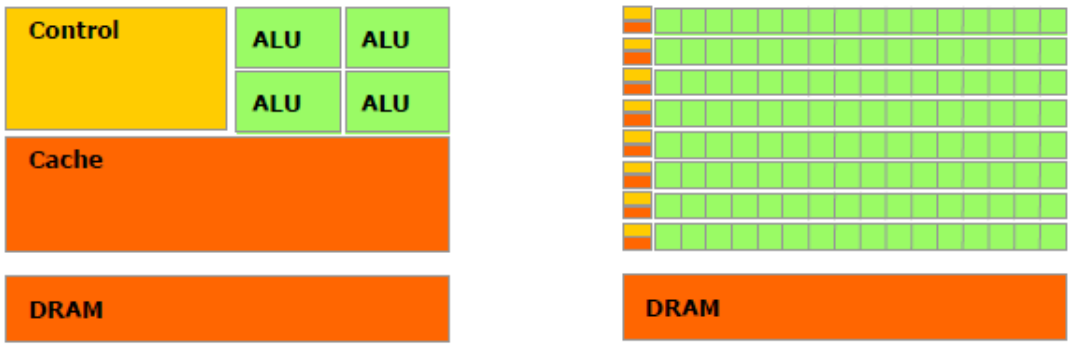
(a) Peak Performance (b) Bandwidth

Figure 2.1: GPU and CPU comparison (taken from [18])

The main reason for the performance difference between a GPU and a CPU is their different design philosophies. While the design of the CPU is based upon a complex control logic, to optimize the execution of a single thread and reduce memory access latency, which includes techniques such as: speculative execution, branch prediction, superscalar, out-of-order

[1] In this context CPU refers to chip unicores and chip multicores

execution and prefetching; the design of the GPU focuses on providing huge computational capabilities and memory bandwidth to optimize the execution of several threads and hide memory access latency. These different philosophies result in different transistor utilization, as shown in Figure 2.2.



(a) On the CPU a good amount of transistors is dedicated to control logic and cache

(b) On the GPU most of the transistors are dedicated to provide computational power

Figure 2.2: Transistor utilization for CPU and GPU (taken from [18])

Besides the differences in their design philosophies, CMPs and GPUs follow different multiprocessing paradigms. A CMP is essentially a MIMD (Multiple Instruction Multiple Data) machine, where each core independently executes a different set of instructions on different data sets, whereas a GPU is a SIMD (Single Instruction Multiple Data) machine, where each core executes the same instructions on different data sets. GPUs are SIMD machines because this paradigm fits the reality of graphics processing where the same operation (set of instructions) is executed on a set of primitives (vertices, lines, triangles).

Although the biggest GPU manufacturers are Nvidia and ATI, this document focuses only on the Nvidia GPUs since they are used in our experimental implementations.

2.2.2 Architecture of the NVIDIA GPU Family

Since its introduction in 1999, the GPU has evolved from a fixed-function pipeline to a fully programmable pipeline.

The first GPU introduced featured a classic pipeline with five major stages: (i) vertex

shader stage, (ii) setup stage, (iii) pixel shader stage, (iv) raster operations (ROP) stage, and (v) memory stage. The main feature of this pipeline is that every stage of the pipeline implements specialized hardware to boost the overall performance. Because the number of pixels operations far exceeds the number of vertex operations, the GPU implements more pixel processors than vertex processors in a ratio of three to one. However, the main drawback of this approach is that depending on the type of application, resources may be insufficient for one stage but sit idle for another stage in the same application. This is because different applications have different shader processing requirements, for instance some application are more pixel-shader intensive while others are more vertex-shader intensive.

The GPU Tesla series was designed with the objective of executing vertex and pixel shader operations on the same unified processor architecture. Shader unification would enable (i) dynamic load balancing of pixel and vertex intensive applications among the shaders of the classical pipeline design, (ii) introduction of new graphics shader stages and, (iii) design of a generic and fast processor. This unified design opens the door to new parallel-computing capabilities to GPUs, but at the same time increases the design complexity.

To satisfy the computational requirements of different graphics applications, modern GPUs integrate hundreds of cores. The Nvidia Fermi family integrates up to 512 cores in a single chip providing a peak performance of 1,331 GFlops for single precision and 665 GFlops for double precision [19].

Figure 2.3 shows the architecture of the NVIDIA Tesla C2070, which is composed of an array of *Scalable Processors* (SPs). The SPs provide all the computation capabilities of the GPU and support IEEE-754 floating-point precision.

The Nvidia Tesla GPU shown in Figure 2.3 has 448 SP cores (or simply cores) grouped into 14 *Streaming Multiprocessors* (SMs). The SM is a computing multiprocessor that executes graphics shader programs and general purpose computing programs. Each SM has 32 cores, four Special Function Units (SFUs), two warp schedulers, two dispatch units, a configurable L1 cache/shared memory and a L2 cache memory. The shared memory acts as a software managed cache, allowing the programmer to use this cache to improve the execution performance. The SFUs are used for transcendental operations and contain four floating-point multipliers.

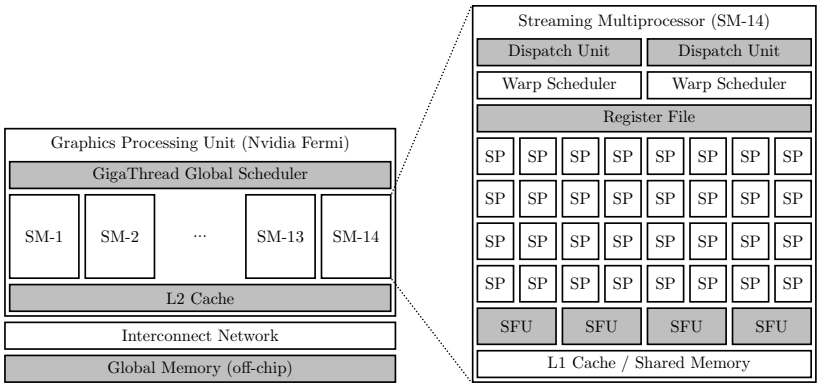


Figure 2.3: Architecture of the NVIDIA Tesla C2070 GPU

Organizing the cores into SMs allows flexibility for providing different levels of performance by having more or less SM in a GPU.

The GPU presented in Figure 2.3 implements an off-chip DRAM memory, which has much higher latency than the shared memory on the SM. Programs should be written in a way that the accesses to this DRAM memory are reduced to achieve higher performance.

Next, for comparison purposes the main features of the NVidia Testla T10 and the C2070 (Fermi), which are used for our implementations, are summarized in Table 2.1.

Table 2.1: NVidia Tesla T10 and C2070 Technical Specifications

Parameter	**Tesla T10**	**Tesla C2070**
Number of Cores	240	448
Clock Speed	1.296 GHz	1.147 GHz
Number of SM	30	14
Number of cores per SM	8	32
Number of 32-bit Registers per SM	16 K	32 K
Shared Memory per SM	16 KB	16 KB / 48 KB
L1 Cache	N.A.	48 KB / 16 KB
L2 Cache	N.A.	768KB
Peak Memory Bandwidth	102 GB/s	144 GB/s
Peak Single Precision Performance	933 GFlops	1.03 TFlops
Peak Double Precision Performance	78 GFlops	515 GFlops
Warp Schedulers per SM	1	2
Dispatch Units per SM	1	2

Besides the differences shown in Table 2.1, each SP core in the Tesla T10 contains a multiply-add (MAD) unit which can issue three floating point operations per cycle, whereas the Tesla C2070 contains a floating point unit that can issue two floating operations per cycle (dual issued pipeline).

2.3 General-Purpose Programming on GPUs

In addition to the tremendous computational capabilities of the GPU, the development of GPUs with fully programable pipelines makes them amenable for accelerating massively-parallel general purpose applications, opening a new application field to GPUs called *General-Purpose Computation on GPUs* (GPGPU).

The availability of GPGPU programming languages, such as CUDA (Computing Unified Design Architecture) and OpenCL (Open Computing Language), have consolidated the adoption of GPU technology as an alternative for accelerating general-purpose applications.

CUDA is a proprietary programming language developed by Nvidia targeting its GPUs. OpenCL is a standard developed by the Khronos group targeting parallel platforms such as Chip Multi-Processors, Digital Signal Processors (DSPs), FPGAs, etc.

Since OpenCL is based on CUDA, the CUDA programming language shares several conceptual foundations with OpenCL; they have similar platform, execution, memory and programming models [20, 21, 22].

2.3.1 Platform Model

The platform model describes a high level representation of the system composed of a *host* and several *devices*. The host orchestrates the program execution on the devices and interacts with the non-CUDA (or non-OpenCL) parts of the program. The devices execute the parallel threads, and they can be a GPU, Chip Multi-Processor, Digital Signal Processor (DSP), FPGA, etc. A device is divided into Streaming Multiprocessors (or Compute Units) and, as explained in the previous Section, a SM is composed of Scalable Processors (or Processing Elements).

2.3.2 Execution Model

A CUDA/OpenCL application is composed of a *host program* and one or more *kernel functions* (or simply kernels), which are executed on the host and on the devices, respectively.

The host program issues a command for scheduling for execution a kernel. When the kernel is executed, the runtime generates a large number of threads (or work-items), organized in a grid, that exploits data parallelism. Hence, a kernel defines the code to be executed by all the threads in parallel when it is executed.

The threads in a grid are organized in a two-level hierarchy. In the top level, the grid has a two dimensional set of thread blocks (or work-groups) and at the lower level each thread block is composed by a three dimensional array of threads, where all the thread blocks have the same number of threads. Thread blocks are assigned for execution to SMs, hence, threads that are executed in the same thread block can collaborate and synchronize among them. Each thread block is assigned an unique two dimensional coordinate that is used to identify uniquely a thread block. In the same way, the threads are assigned a unique three dimensional coordinate that identifies uniquely a thread.

The size of the grid and the thread blocks are passed as parameters when the kernel is scheduled for execution.

The execution model does not assume a particular execution order of the thread blocks, and, hence, it does not support synchronization among thread blocks to avoid potential deadlocks.

2.3.3 Memory Model

The data required by the kernel should be transferred from the host memory to the device global memory. Although having a large number of threads hides the memory latency, it imposes a large traffic in the interconnect network to global memory which might increase the memory latency.

In order to minimize the traffic in the interconnect network, a device provides additional types of memory with different latencies:

- Global Memory, it can be read or write by any thread. It is the largest on the GPU and

the slowest, therefore, it should be efficiently used. In some GPU families global memory is cached,

- Constant Memory, it resides in global memory and remains constant throughout the execution of the kernel,
- Texture Memory, it is only defined by CUDA and it refers to a global memory area that it is cached. The cache is optimized for 2D spatial locality,
- Shared Memory (Local Memory), this memory area is shared by the thread block and, hence, shared by the threads in the same thread block,
- Registers (Private Memory), are the fastest memory on the GPU. This area is private to a thread, and
- Local Memory, only defined by CUDA, it is a memory area private to a thread that resides in global memory. In general it is used by automatic variables that cannot fit in the registers.

2.3.4 Programming Model

CUDA and OpenCL supports two programming models: data and task parallelism. Data parallelism is the philosophy design behind CUDA kernels, i.e. the same task executed in different pieces of data. Task parallelism refers to executing different tasks in parallel, which is achieved by executing concurrently different tasks on the host and on the device or on different devices. Some GPU families provide support for executing different kernels in parallel in the same GPU [19].

2.3.5 Thread Scheduling

The GigaThread scheduler assigns threads for executing to SMs depending on the number and size of the thread blocks defined at the time the kernel was sent for execution.

The number of thread blocks assigned to a thread block depends on the maximum number of thread blocks that a SM can support, as long as no resources are exceeded such as shared

memory or registers. The Nvidia Tesla T10 supports up to eight thread blocks assigned with at most 2KB of shared memory per thread block, however, if each thread block needs 2.1KB then only seven thread blocks can be allocated per SM.

For performance purposes the threads of a thread block are executed in *warps*. A warp is a set of 32 threads that are executed in lock-step mode one instruction at the time. When an instruction involves a long latency, i.e. memory, operation, the scheduler switches the execution to a ready warp; this allows to hide long latency operations. When the instruction finishes its long latency operation, waits until it is scheduled for execution again. Similarly, to hide the ALU latencies, instruction from several warps are interleaved and sent to the pipeline for execution. In this context, selecting a ready warp does not incur any delays, which is known as zero-overhead thread scheduling.

As it can be noticed, a GPU can hide both ALU and memory latency by running multiple warps per SM. The idea is that the scheduler has many warps to choose from and send for execution and by this way to minimize the performance loss due to latency.

2.4 Executing Applications on Multi-GPU Systems

Multi-GPU systems are becoming popular not only for achieving higher levels of parallelization, but also for overcoming the memory size limitation imposed by single GPU systems [23, 24, 25, 26, 27].

Performance on multi-GPU systems has been studied in works by Schaa et al. [23] and Spampinato et al. [24]. The objective of these works is to provide accurate analytical models for predicting the execution time, taking into account computation time as well as network and PCI-e interconnect communication time. These models show that communication between GPUs has a negative impact on the execution time for applications where the data communication time is bigger than the computation time. The accuracy of these analytical models is experimentally tested with benchmarks running on actual muti-GPU systems, and it is shown that the predicted times are within 11% to 40% of the actual measured execution time.

Running single-GPU applications on multi-GPU systems can be approached by either providing libraries that parallelize a single GPU application at runtime [25, 26] or instrumenting

the application before compiling [27].

In [25] a framework for exploiting higher levels of parallelism in multi-GPU systems and computer clusters is proposed and implemented. This framework, called CUDASA, is composed of four layers: application, network, bus and GPU layer, see Figure 2.4.

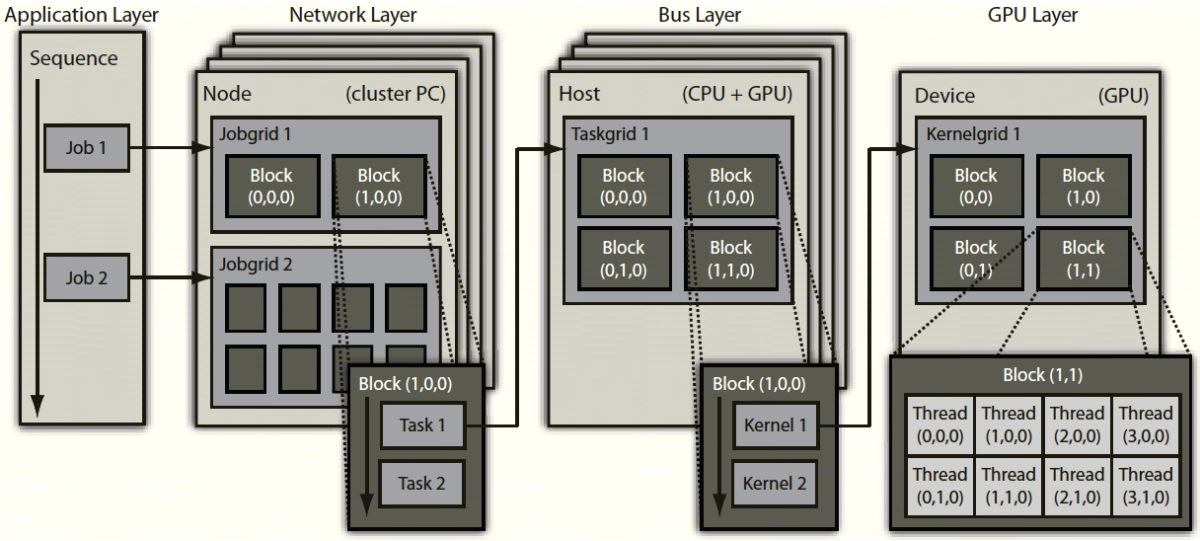


Figure 2.4: Overview of the four abstraction layers of CUDASA (taken from [25])

The three lower layers exploit parallelism at different hardware levels, i.e. GPU, multiple GPUs in a single system and at the computers that compose the cluster; while the topmost layer contains the sequential function calls that exploit the parallelism provided by the underlying layers. To provide extended levels of parallelism this framework has three components: (i) a runtime library, which provides the basic functionality for job and task scheduling, distributed shared memory management and common interface functions, (ii) a set of language extensions that introduces function interfaces and type qualifiers for the higher layers of abstraction, leaving the GPU layer unchanged, and (iii) a self-contained compiler that acts as a CUDA pre-compiler. The language extensions of CUDASA allow to specify the type of subroutine, i.e. job or task, and the size in a similar way the size of a kernel is defined in CUDA. The libraries keep internal queues with blocks ready for execution and distribute the pending blocks to GPUs. Although the authors present good scalability results for dense matrix multiplication for up to 4 GPUs in a single system, for computer cluster configurations the scalability is fairly poor due to the high communication overhead.

Kim et al. [27] present a framework that decomposes at runtime an OpenCL kernel written for a single GPU into multiple CUDA kernels and executes them on multiple GPUs. The framework is composed of three components: (i) a command scheduler that dequeues each command queued in the original OpenCL program and schedules the dequeued commands in the ready-queue, (ii) a virtual GPU device that provides the single compute device image through a set of assignment queues, and a virtual buffer, which contains all the processed commands, and a master copy of the actual device buffers, respectively, and (iii) a CUDA runtime, which defines as many CPU threads as GPUs the system has and executes the commands in the assignment-queues, see Figure 2.5. The workload is partitioned such that the data transfer is minimized for each partition, which is achieved by performing access range analysis of the arrays accessed by the kernel.

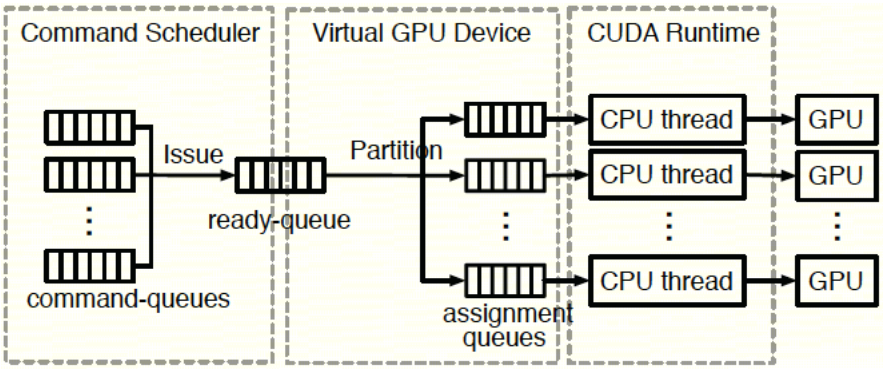


Figure 2.5: Framework for achieving a single device image (taken from [27])

The speedups presented for a set of applications show linear speedup for the majority of the applications, while for few applications show no linear speedup or no speedup at all is achieved, which according to the authors is due to the uneven workload distribution and false data sharing among partitions.

Chen et al. [28] present a framework for execution of CUDA applications on multiple GPUs that exploits fine-grained tasks to achieve efficient load balance in the system. This framework defines: (i) a global task container that has all the tasks that are to be executed, (ii) a local task container that it is associated to a device, (iii) one host process per device, and (iv) a persistent kernel launched at the beginning of the computation on each device. The host process moves tasks from the global task container to the local task container when there is

free space on it and informs the device. The kernel running on the device fetches tasks from the local task container and executes them by groups of threads, called *tasks execution units* (TEUs). Multiples TEUs can be executed concurrently on the same device, which, according to the authors, can be more efficient than executing one TEU on the whole device when there is limited data parallelism which can be handled by a few threads. Two granularity levels are proposed: thread block level and warp level. The framework is evaluated on a system with four GPUs; for uniform workloads, the warp-level granularity presents some imbalance and execution overhead compared to the static scheduling and tread block level granularity, for non-uniform workloads the static scheduling presents poor load balancing, while the thread block and warp granularities level present good load balancing, with the warp-level granularity performing slightly better. Although the results presented are encouraging, no information about speedup achieved or overhead is presented.

StreamIt [29, 30] is a platform-independent programming language designed to expose the parallelism and communication of streaming applications. StreamIt provides basics constructs for representing application parallelism and communication without depending on the topology or computation granularity of the underlying architecture. Hagiescu et al. [31] propose an automated compilation flow for mapping StreamIt programs onto GPUs. An scalable mapping framework that extends the work in [31] to multi-GPU systems is proposed in [32]. This framework has four components: (i) an algorithm for partitioning the StreamIt application, (ii) a global mapper that balances the partitions among the available GPUs, (iii) a code generator for producing the code for the partitions and communications among them, and (iv) an execution environment that provides a controller that coordinate kernels loaded on each CPU and an inter-partition memory communication scheme for pipelined execution. This framework is experimentally compared to the single GPU implementation, in general the results present a reasonable speedup for systems with two and three GPUs, however, for four GPUs the speedup is diminished possibly due to the data communication.

The frameworks described above provide support for multi-GPU execution of single-GPU programs through an external library, which reduces their portability and performance. On the other hand, instrumenting the application for providing support for execution on multi-GPU

systems does not require additional libraries, hence, no extra overhead is included and better performance can be achieved.

Instrumenting an application can be approached as a manual or automatic process. Manual instrumentation has the advantage that fine tuning can be achieved, but on the other side it is complex and prone to error.

In the domain of manual instrumentation, several works present multi-GPU implementations of dense algebra operations [33], linear programming [34], optimization problems [35], computational fluid dynamics [36], medical imaging [37, 38], signal processing [39], among others. All these works apply techniques for achieving fine tuned implementations of their problems.

Automatic instrumentation for supporting execution on multi-GPU systems involves source to source conversion, i.e. convert an application written for single-GPU systems into an application for systems with multiple GPUs. At the time of this writing no works for automatic source to source conversion for executing applications written for single-GPU systems on multi-GPU systems have been found.

2.5 Failures and Fault Tolerance

With the current microprocessor fabrication trends, i.e. smaller feature sizes, lower voltages and faster clock frequencies, microprocessors are becoming increasingly susceptible to hardware failures [40, 41]. Hardware failures can be classified as: transient (soft) failures and permanent (hard) failures.

Microprocessors can be protected against failures by implementing some form of redundancy. Thus, when a failure occurs, it is masked by the redundancy, keeping the microprocessor functioning as though the failure did not take place. Hardware redundancy can be achieved in three different ways:

- *Spatial redundancy* is achieved by executing the same instructions on multiple independent functional units at the same time.

- *Temporal (Time) redundancy* is achieved by carrying out the same computation multiple times on the same functional units at different times; and
- *Information redundancy* is achieved by adding extra data (bits) to the information used in/computed from computations.

Next, transient and permanent failures and the techniques used to mitigate them are described.

2.5.1 Transient (Soft) Failures

Transient failures cause a component to malfunction for a brief period of time, after which the functionality of the component is fully restored. Transient failures are caused by alpha-particles in the chip material, cosmic rays from space, or radiation from radioactive reactive atoms [42]. Transient failure rate is increased by the effects of transistor integration, with lower threshold voltages and capacitances, smaller charges (Q=CxV) are needed to flip a bit in memory or in the datapath or control logic [42, 43]. Typically, a microprocessor has a Soft Error Rate (SER) of 4000 FIT[2], where 50% affect the datapath and control logic, and 50% affect the on-chip memories.

2.5.1.1 Fault Tolerant Techniques for Memory Structures

Data in memory can be protected against bit flips through information redundancy. The most common form of information redundancy is *coding*, which adds extra bits to the data stored. Extra bits allow to detect or to correct errors before using the data. Parity bits and ECC (error-correcting code) are the coding schemes most commonly used for detecting and correcting errors. In general, DRAM (dynamic RAM), L3 and L2 caches are ECC protected, whereas L1 cache memory is parity protected. Standard ECC algorithms can automatically correct single-bit errors and detect double-bit errors. ECC is reported to correct up 90% of single-bit errors [44].

[2]FIT (Failures In Time): Errors per billion (10^9) hours of use.

2.5.1.2 Fault Tolerant Techniques for Datapath and Control Logic

Several techniques based on spatial, information and temporal redundancy have been developed to protect the datapath and the control logic against transient failures. These techniques show a trade off between failure protection coverage and additional logic needed, i.e. area and power overhead; for instance they can protect all the datapath and control logic, only the datapath, or only a subset of the datapath.

Spatial Redundancy Techniques. The most commonly spatial redundacy-based technique used is the *replication check*, which, depending on the application, duplicates (DMR: Dual Modular Redundancy) or triplicates (TMR: Triple Modular Redundancy) computational logic for error detection or correction, respectively. Besides replicated computational logic, a comparator or voter is needed to detect or correct, respectively, transient failures. The IBM S/390 G5 processor achieves 100% error detection from any transient error on the datapath through complete duplication of the instruction and execution units. Duplicated instruction and execution units are arranged in parallel to, and operate in lock-step with, their correspondent primary instruction and execution units. To guarantee correct execution and recoverability, a special unit, the *Recovery Unit (RU)*, is used to detect errors, through output comparison. If the RU finds no errors, it checkpoints the microarchitectural state and the register file into an ECC protected checkpoint array every time an instruction completes; in the case of erroneous execution, the execution is restarted from the last stored checkpoint [45]. The area overhead paid by the G5 processor for this added reliability is, according to Spainhower and Gregg, about 35% [46]. Figure 2.6 shows a high level view of the architecture of the IBM G5 processor.

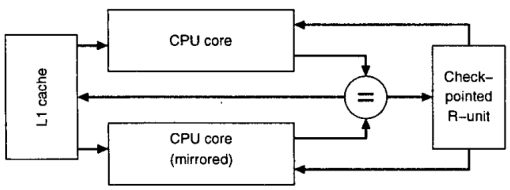


Figure 2.6: High level view of the IBM G5 processor (taken from [47])

Although full duplication of the datapath and control logic provides robust failure protection

and easy implementation, it inefficiently utilizes the resources. The inefficiency arises from the fact that both datapaths:

i) Execute all the instructions irrespective of their usefulness, which is necessary to keep both datapaths synchronized, i.e. running in lock-step; and

ii) Stall when execution finds cache misses or data dependencies.

Indeed, only non-speculative instructions need to be verified, therefore, re-executed; and not both executions need to stall because of cache misses. Based on these premises, robust transient failure protection can be achieved without full duplication of the datapath.

Selective Series Duplex (SSD) provides failure protection at a lower cost in terms of computational logic replication [47]. Savings are achieved by selectively replicating some parts of the computational logic, i.e. the pipeline, avoiding unnecessary front-end logic replication, such as: branch prediction hardware, prefetching hardware, reorder buffer, among others. This is possible because the two pipelines work in series, instead of in parallel, therefore, by the time completed instructions reach the redudant or verification pipeline, V-Pipeline, branch decisions and cache misses are already cleared, and hence no additional hardware for performing those tasks is needed. A FIFO buffer, called *the Re-Execution Queue (REQ)*, is utilized to store the instructions, along with their results, that are ready to be commited by the main pipeline, i.e the P-Pipeline, and ready to be executed by the V-Pipeline. Instructions are released from the REQ buffer once they are executed by the V-pipeline. The P-Pipeline can continue executing instructions as long as the REQ buffer has space to allocate completed instructions. When the REQ buffer is full, the P-Pipeline starts to stall, which increases the execution time. Depending on the number of instructions that can be issued and commited per cycle by the P-Pipeline and by the V-Pipeline, SSD increases the average execution time up to 1.3%, and achieves a reduction of up to 61% in the area overhead compared to the case of full datapath and control logic duplication. Figure 2.7 shows the organization of the main components of the SSD architecture.

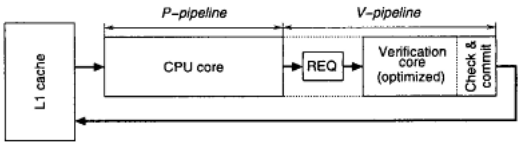


Figure 2.7: High level view of the SSD architecture (taken from [47])

Information Redundancy Techniques. As with memory, information redundancy can be implemented to protect the datapath and control logic against transient faults. The IBM Power6 microprocessor [48] includes error detection mechanisms for most of its control logic. Residue checking is used in the floating-point units, parity protection is used for most of the latches in the data-flow circuits, and logical consistency checkers, which check that states are valid with respect to their state machine, are used to protect the control circuits. In the same way as the G5, the Power6 utilizes a Recovery Unit (RU) to checkpoint the system state into an ECC protected checkpoint array, and to restart the execution from the last checkpoint stored, if a failure is detected.

Temporal Redundancy Techniques. Unlike, spatial and information redundancy implementations that add hardware overhead, implementations based only on temporal redundancy do not involve adding hardware with re-execution purposes. However, carrying out the same computations on the same functional units several times increases the average execution time. There are two main sources for increasing execution time:

1. *Separation between original and redundant executions.* To be able to detect transient failures with duration Δt, multiple executions should be separated by a time period greater than Δt. Therefore, selecting the time separation among the original and redundant executions should be done carefully because, while a long delay results on delayed instruction committing, a short delay results on inability for transient failure detection.

2. *Redundant execution itself.* When an instruction is re-executed using resources on the pipeline, no other instructions, that need those busy resources, can be executed until the resources are released.

As it can be seen, temporal redundancy imposes trade-offs between the execution time overhead and the transient failure protection coverage.

REESE (REdundant Execution using Spare Elements) [49] implements time redundancy, exploiting idle periods on functional units for transient failure detection. The premise for this scheme is that on modern microprocessors idle periods account for about 30% to 40% of the total available time. REESE re-executes instructions during idle periods, achieving decreased execution time overhead compared to the one imposed by naive time redundancy schemes. Instructions are grouped for execution, in such a way that groups of new instructions, called P-Streams, are interleaved with groups of re-execution instructions, called R-Streams. Interleaving and grouping should be designed aiming to maximize ILP.

A FIFO buffer, called R-Stream Queue (RQ), is implemented to store P-Stream instructions, along with their results, that are ready to be committed, that is, ready to be re-executed. Additional scheduling logic is needed to interleave P-Stream instructions and R-Stream instructions. To decide whether P-Stream or R-Stream instructions should be executed, the scheduler takes into consideration data and control dependency conflicts found on the P-Stream and the state of the RQ. Dependency conflicts as well as a full RQ stall the processor until dependencies are cleared out or space is freed in the RQ. In general, if no dependencies among P-Stream instructions are found and space is available on the RQ, P-Stream instructions are executed, otherwise R-Stream instructions are executed. REESE achieves 14% performance decrease when compared to a microprocessor with no time redundancy.

2.5.2 Permanent (Hard) Failures

Hard failures are directly related to physical damages on the transistor. The transistor has four terminals, where the gate terminal controls whether a channel to connect the n-doped silicon drain and source is created; the fourth terminal (body) is connected to ground in Positive-Channel Metal Oxide Semiconductors (NMOSs) or to V_{DD} in Positive-Channel Metal Oxide Semiconductors (PMOSs). The oxide gate is a very thin insulating layer of SiO_2 that prevents current flow from the gate to the body. Figure 2.8 shows the cross section of an NMOS transistor.

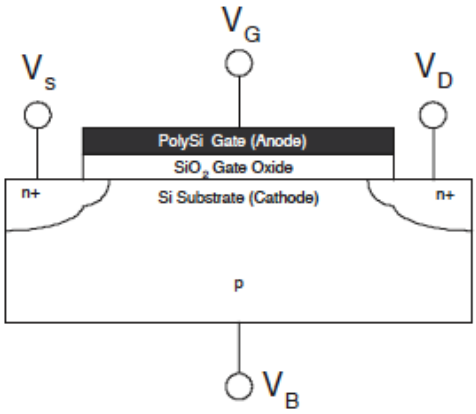


Figure 2.8: Cross section of an NMOS transistor (taken from [50])

2.5.2.1 Types of Permanent Failures

Permanent failures are mainly due to process variation, material defects, and physical failures during use (operational hard failures), and causes a component to malfunction permanently. As it will be seen in the next paragraphs, permanent failure rate is affected negatively by technology scaling.

Most of the material defects are detected by quality control tests before shipment, the remaining undetected material defects often manifest as failures in a similar fashion as the ones caused by process variation, therefore, material defect related failures are not discussed in detail in this document.

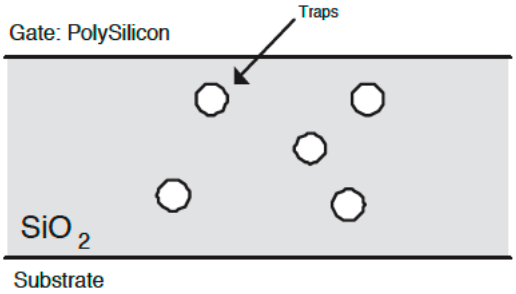
Process Variation. Although the microprocessor fabrication process is done with highly precise equipment and in impurity isolated environments, microprocessors present variations in film thickness, lateral dimensions, oxide thickness, and doping concentrations. Oxide thickness, and doping concentration variation are becoming more prominent as the microprocessors approach the nanotechnology, inducing variations on the threshold voltage. In [51], it is shown through simulations that as the channel length (feature size) decreases, the threshold voltage variation increases from around 40mV to 100mV.

Considering that for 35nm MOSFET generation the desirable threshold is 100mV to 200mV and a supply voltage below 1V, these results show that small channel lengths are prone to *suck-at-x faults*, potentially leading to permanent failures.

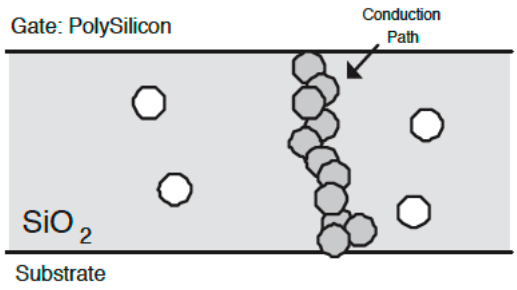
Operational Hard Failures. Technology scaling reduces feature sizes and voltage levels of transistors. In an ideal scaling scenario, every new generation, the number of transistors on a chip doubles, the dynamic power per transistor decreases by about 50%, and the power remains unchanged. However, in practice power and consequently the temperature have increased at an alarming rate, which affects the lifetime reliability of the microprocessor.

Simulations show that for a given microarchitectural pipeline, shrinking the feature size from 180nm to 65nm, results in the temperature increasing by 15 degrees Kelvin, and the failure rate increasing by 316% [52]. Gate oxide breakdown (ODB) and electromigration are likely to be the most dominant phenomena that cause operational hard failures [52, 53].

Gate oxide breakdown (ODB) results in the malfunction of a transistor due to the creation of a conduction path through traps[3] in the gate-oxide from the polysilicon gate (anode) to the substrate (cathode) [50]. Traps are present in newly manufactured oxide due to imperfections in the fabrication process. Over the operational time, more traps can appear as a result of electric field stress, and eventually several traps can line to form a conduction path. The ODB rate increases as the oxide gets thinner and the temperature increases. Figures 2.9a and 2.9b show traps and a conduction path, respectively, formed on the gate oxide.



(a) Formation of traps in the gate oxide



(b) A conduction path created by traps in the gate oxide

Figure 2.9: Gate oxide breakdown (ODB) (taken from [50])

Electromigration is the gradual displacement of the metal atoms of a conductor as a result of high current densities flowing through that conductor. In microprocessors, electromigration

[3]Traps are defects within the gate oxide. They are called traps because those defects can trap charges.

causes wear-out of the interconnects, leading to open circuits, which is aggravated by smaller wire lengths, present in device scaling, and high operating temperatures.

2.5.2.2 Fault Tolerant Techniques for Permanent Failures

Fault tolerance in the presence of hard failures can be achieved with hardware or software techniques.

Hardware Techniques. Hardware techniques aim for non-stopping operation of the microprocessor and memory components. Depending on the hardware implementation, non-stop operation and degraded operation, until the failed hardware is replaced and normal operation is restarted, is allowed. Hardware implementations include: replication check techniques, memory and CPU sparing. Hardware sparing consists of having spare hardware ready to replace similar pieces of hardware in case of permanent failures. Sparing can be classified into: hot and cold sparing. Hot sparing provides non-stop operation because the spare hardware is always running and has the same architectural and microarchitectural state. Cold sparing requires stopping the operation and restarting from a consistent state previously saved. Evidently hot sparing does not impose execution time overhead, but it requires redundant hardware. Cold sparing makes more efficient use of hardware resources, spare hardware can be utilized for normal execution, and when a hard failure is detected, the spare hardware continues the execution that was being done by the failed hardware.

Although replication check techniques and hot sparing allow for non-degraded continuous operation in the presence of hard failures, power and area overhead increases by two or three times. Overhead is mainly due to the fact that synchronization is required, which increases the control logic complexity.

Redundant Multithreading (RMT) is a technique that, similar to replication check techniques, runs redundant threads of the program and compares their outputs to detect failures. RMT provides tolerance to transient faults when implemented on unicore multi-threaded processors, and tolerance to hard faults when implemented on CMPs, i.e. both threads run on separate hardware.

The main differences between RMT and check techniques are: (i) RMT does not require redundant lock step execution, allowing for less hardware overhead, and (ii) output comparison is done at instruction level rather than at cycle level, which means that redundant hardware only needs to have the same architectural state, not the same microarchitectural state.

Chip-level Redundant Threading (CRT) implements RMT on a dual-core microprocessor [54]. CRT relaxes the synchronization constraint, allowing the redundant thread, i.e. trailing thread, to run sufficiently behind the original thread, i.e. leading thread, to avoid stalling and performance overhead due to cache misses and branch miss-predictions, respectively. CRT replicates the inputs and compares the outputs at the instruction level, as it can be seen in Figure 2.10. Input replication allow both threads to read the same data, which introduces complexity because the trailing thread runs behind, and therefore could read updated data.

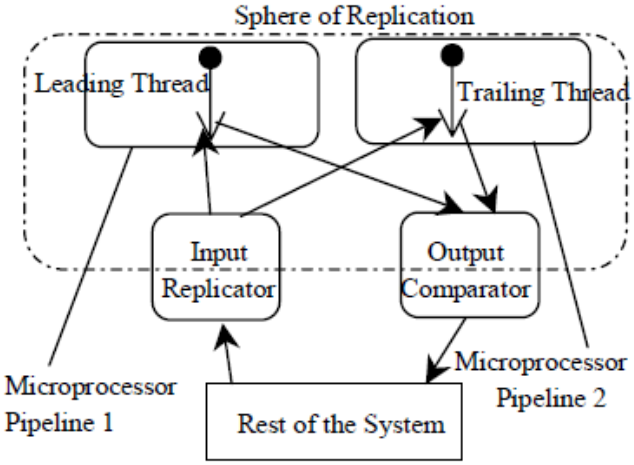


Figure 2.10: Chip-level Redundant Threading (CRT) (taken from [54])

The Branch Outcome Queue (BOQ) and Load Value Queue (LVQ) are two special structures implemented by CRT to avoid stalling and branch miss-predictions on the trailing execution. Ideally the trailing execution neither stalls nor has branch miss-predictions, however, in reality this depends on the lag between the leading and trailing executions. On single thread applications, performance in CRT is the same as in DMR, but for multi-thread applications, CRT outperforms DMR by 13% on average.

Dynamic Implementation Verification Architecture (DIVA) [55] splits the traditional microprocessor into the speculative DIVA core and the DIVA checker, see Figure 2.11. The DIVA

core is an out-of-order execution microprocessor that has all the functionality to issue, decode, execute and store the results in the reorder buffer, but it cannot commit instructions. The DIVA checker has a functional checker stage that checks the correctness of the computation (CHK), if there are no errors the results are passed to the commit stage (CT), else the checker fixes the errors, flushes the DIVA pipeline and restarts the execution at the next instruction. The checker is assumed to be failure free through ECC protection and there are no communication errors, hence, all the data DIVA reads or writes to memory or register is correct. Besides, the DIVA checker is organized into two indepent parallel pipelines: the CHKcomp pipeline verifies the correctness of all computations, and the CHKcomm verifies all the communications with the core are correct. According to simulations, in the absence of faults, DIVA performance overhead accounts for about 3%.

A watchdog timer and a counter are implemented to detect total failures of the core. The watchdog timer detects when the core times out before retiring an instruction, and increases the counter; once the counter reaches a threshold, the checker takes over the execution and continues with degraded performance, providing fault tolerance to hard failures.

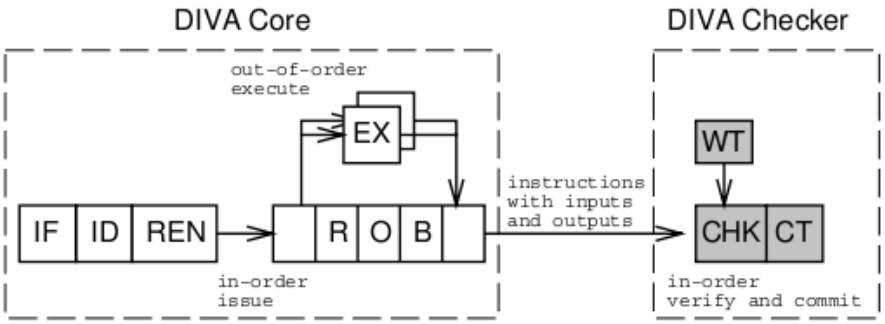


Figure 2.11: Dynamic Implementation Verification Architecture (DIVA) (taken from [55])

Software Techniques. Indeed, hardware techniques effectively provide non-stop fault tolerance to hard errors; however, in order to achieve failure resilience and to avoid high performance overhead, all of them impose area overhead. This is reasonable for critical systems that require both high performance and availability at the same time; however, for commercial-grade systems, adding specialized hardware for fault tolerance purposes is not an option due to high costs. Off-the-shelf microprocessors, utilized in comercial-grade systems, implement basic fault

tolerant techniques, such as: parity and ECC protection for cache and main memory.

Off-the-shelf microprocessors have been used as the building block for high performance systems, the latest top 500 supercomputer list released on November 2012 [56] shows that the top supercomputers are build out of commercial microprocessors; for instance: the Titan at the Oak Ridge National Laboratory, ranked first at the time of this writing, is composed of 18,688 nodes, each with a 16-core AMD Opteron 6274 processor and an NVIDIA Tesla K20X graphics processing unit (GPU) accelerator [57, 58], and the TH-1A at the National Supercomputer Center in Tianjin, ranked eight at the time of this writing, is composed of 14,336 Intel Xeon X5670 six-core processors [59, 60].

Fault tolerance for off-the-shelf microprocessors is implemented through software techniques, which do not need additional hardware to support them. However, software techniques impose higher performance overhead compared to hardware techniques.

Software techniques can be classified as: reactive and proactive policies. Reactive fault tolerant policies minimize the effects of the faults on the execution of applications when the fault occurs. Proactive fault tolerant policies predict faults and take actions to avoid an application crash [61, 62, 63]. Failure prediction is based on information collected by the system about the health of the system.

Among reactive policies, checkpointing/restart-based error recovery is the most commonly used [15, 64, 65, 66, 67, 68]. Depending on the implementation, either the application itself or the operating system periodically save the system/application state in a checkpoint. When the application crashes as a result of a failure, the most recent checkpoint is used to restart the application from the state saved in the last successful checkpoint. Therefore, checkpoints save time by restarting the application from a saved point instead of restarting from the beginning.

In terms of performance, the proactive approach has shown to be more efficient than the reactive approach. However, it is not always possible to predict failures, some studies have shown that up to 76% of failures can be predicted. Besides, upon failure prediction, an action has to be taken, for example: checkpoint the system or application.

2.6 The Checkpoint/Restart Technique

As mentioned in the previous section, the Checkpoint/Restart technique is the most commonly used to achieve fault tolerance in the presence of permanent faults. Under this approach, the process state is saved periodically into stable storage, i.e. checkpointed. The checkpoint period, the time between two consecutive checkpoints, has direct effect on the running time of the application [69, 70, 71].

The checkpoint/restart technique offers fail-stop protection, which means that the application stops after a fault occurs and has to be restarted.

Checkpointing can be implemented in either of the following three ways: by the application itself, through an external library, or by the operating system [13, 14].

Depending on the implementation, different issues arise: storage efficiency, transparency and restart technique. Storage efficiency refers to whether only the required structures or more than the required structures are saved in the checkpoint. Transparency is achieved when no changes need to be made to the application code. Restart techniques can be application dependent or independent (common); a common restart technique can be executed by an automated system, whereas an application dependent technique requires a specific script for restarting the application from a checkpoint. Table 2.2 summarizes the main features of each scheme:

Table 2.2: Features Summary for Checkpointing Schemes

	Checkpointing Scheme		
Feature	**Application**	**Library**	**O.S. kernel**
Storage Efficiency	High	Low	Low
Transparency	Low	Medium	High
Machine & O.S.-Specific	No	Yes	Yes
Restart Technique	Application Dependent	Common	Common

There are three measures of checkpoint performance:

1. Checkpoint Latency, refers to the time that it takes to save the application/system data.

2. Checkpoint Overhead, is the time added to the application execution time as a result of checkpointing.

3. Checkpoint Size, is the average size of the checkpoint.

The main goal of a checkpoint implementation is to minimize the three measures. However, most of the times checkpoint implementations aim for minimizing the checkpoint overhead because users prefer to risk running applications without checkpointing rather than a checkpointed application with high checkpoint overhead.

Checkpoint overhead increases as the checkpoint period decreases, and vice versa. Under the presence of faults, it is better to checkpoint as often as possible because the amount of work lost would be small. For failure free executions, the less often the application is checkpointed, the lower the checkpoint overhead is. Evidently this suggests a trade-off between checkpoint overhead and checkpoint period.

To minimize the checkpoint measures, checkpointing can be incremental or non-incremental:

1. Incremental Checkpointing saves only the application/system data that has changed since the last checkpoint. To recover a system with incremental checkpointing, all the checkpoint files are needed to leave the application/system in the state that it had at the moment that the checkpoint was taken. To avoid a large number of checkpoint files, after taking a certain number of incremental checkpoints, a non-incremental checkpoint has to be taken and all the previous checkpoint files are deleted.

2. Non-Incremental Checkpointing saves all the application/system data in every checkpoint. Only the last checkpoint file is needed to recover the application/system state to the state it had at the time the checkpoint was taken.

While the incremental approach saves time when the checkpoint is taken by saving only a subset of the data, the non-incremental saves time when the restart/recovery is taking place by having all the information needed in only one file.

2.6.1 Application-Level Checkpointing

In the application-level checkpointing technique, the application developer inserts pieces of code to explicitly save the state of the application. Application-level checkpointing offers the following advantages: (i) highest level of efficiency since the application developer has exact knowledge of which structures and when must be checkpointed, and (ii) hardware and operating system independency. The main drawbacks of application-level checkpointing are: (i) it requires changes to the application code, and (ii) the application imposes constraints on when a checkpoint can be taken.

Cornell Checkpointing Compiler (C^3) is an application level checkpointing implementation for multi-thread applications in shared memory [66]. C^3 has two components: (i) a pre-compiler for source-to-source conversion of the application, and (ii) a runtime system that supports a protocol for coordinating the checkpoint and restart of the application threads. In C^3 the application developer has to insert `potentialCheckpoint()` calls in places where it may be safe to take checkpoints. As the subroutine name suggests, a call to potentialCheckpoint() does not mean that a checkpoint is taken, it depends on how much time has passed since the last checkpoint was taken. The checkpoint is done in three steps: each thread (1) calls a barrier, (2) saves its private state, and (3) calls a second barrier. To avoid deadlocks between threads, they are forced to take a checkpoint as soon as they hit a program barrier and another thread has initiated a checkpoint operation, which is indicated through a flag in shared memory. The portability of C^3 is successfully shown in two different architectures: a two-way Athlon machine running Linux and a four-way Compaq Alphaserver running Tru64 UNIX. On these two architectures, the checkpoint overhead is about 2% to 3% of the non-checkpointed execution.

2.6.2 User-Level Checkpointing

In this approach, user-level libraries are provided to handle the checkpoint/restart operations. Library implementations address the drawbacks suffered by the application-level techniques: (i) it may not require application code changes or at most it requires small changes, and (ii) it does not present constraints for choosing the checkpoint period. However, library imple-

mentations take core-dump-style snapshots of the computational state of the machine, which results in large checkpoint files compared to the checkpoint files obtained by application-level checkpointing implementations.

Libckpt is a checkpoint library implementation for uniprocessor systems running linux, supports incremental and non-incremental checkpointing as well as automatic and user initiated checkpointing. Automatic checkpointing does not require changes on the application, hence, only requires re-linking the object files with the library. User initiated checkpointing allows the user to specify certains places in the application code where it is more advantageous to take checkpoints, hence, it requires calling `checkpoint_here()` and recompiling the application to link the library [64]. Libckpt takes control of the program, generates an interrupt periodically and takes a sequential[4] checkpoint, although libckpt support asynchronous disk transfers[5] as well. Libckpt shows that by using incremental checkpointing, savings of up to 90% in the checkpoint size can be achieved compared to the non-incremental case; in the same way, savings of up to 80% in the checkpoint overhead can be achieved, however for a minority of applications, an increase of up to 20% in the checkpoint overhead is shown.

Condor is a user-level library based checkpoint implementation for distributed processing systems using linux [65]. Similar to libckpt, the application does not need code changes and hence it only requires to link the objects files to the library. Condor implements checkpoint/restart for process migration purposes; the main idea is to migrate processes to idle stations, which achieves better resource utilization. A signal starts the checkpoint process, which saves the process state, open files attributes, signals and processor state. Condor augments several system calls, such as: `open()`, `write()` and `printf()`, to have the ability of saving the information required for checkpointing. Although Condor supports a variety of user applications, it does not support checkpointing of communicating processes.

[4]Sequential checkpoint does not interleave computation with disk transfers.
[5]Asynchronous disk transfer allows interleaved computation with disk transfers.

2.6.3 Kernel-Level Checkpointing

In the kernel-level checkpointing approach, all the functionality is provided by the kernel of the operating system. The highest level of transparency is achieved, since no application code changes and no recompiling is required. However, similarly to the user-level checkpointing approach, kernel-level implementations have large checkpoint sizes, and portability among heterogeneous platforms is not possible.

Kernel-level implementations have different approaches: some of them are implemented as dynamically loadable kernel libraries, as kernel threads, and others as modules embedded in the operating system.

VMADump (Virtual Memory Area Dumper) is a kernel-level checkpointing implementation, similar to Condor it is used for process migration with load balancing capabilities. Since VMADump is used for process migration, it is invoked directly by the process, and hence it is not totally transparent. Once the checkpoint is initiated, process attributes, CPU registers, signal handlers and memory are saved [13].

A similar kernel-level implementation to VMADump is CRAK, except that CRAK is not user-initiated, therefore, it is totally transparent [13, 67]. Although CRAK is classified as a kernel-level implementation, it does not modify the kernel of an existing operating system, but instead it adds a new kernel module. To support different versions of linux, the functionality of CRAK is divided among the user and kernel level of the operating system; the user level is preferred for some tasks because it is more 'standard', and therefore it is easier to do those tasks correctly. The user level is used for identifying the processes to be checkpointed and stopping them, after that the kernel level saves the following information: process credentials, signal handlers, pending signals, file descriptors attached to sockets and regular files.

TICK (Transparent Incremental Checkpointer at Kernel level) is a kernel-level, transparent checkpointing implementation for linux clusters [68]. TICK provides (i) full or incremental checkpointing capabilities, (ii) transparency, (iii) flexibility, since many options can be controled from files on `/proc` and, (iv) high responsiveness, since the checkpoint can be triggered within microseconds.

Checkpointing applications running on clusters, called *Distributed Checkpointing*, presents challenges since the state of the system includes the state of different processes, located on different nodes. A consistent global or system-wide checkpoint is a set of local checkpoints, taken at every node where the application is running, that represents a consistent state of the system at a particular instance of time. Clearly, when processes are communicating among themselves, taking a global checkpoint is not straightforward, and it requires either some degree of synchronization among the processes or a method to determine a global recovery line among a set of uncoordinated checkpoints.

To achieve consistent global checkpoints, TICK needs to provide global synchronization, as well as local support. To provide global synchronization, the *Buffered Coscheduling* (BCS) is utilized; BCS divides the global execution into slices, at every time-slice: new communication calls are buffered until the next time-slice, and communication-pending information is exchanged among nodes. To provide local support, TICK is implemented as kernel threads, where there is one thread per processor. With these two techniques, global recovery lines are automatically established because local checkpoints are taken after the communication calls are buffered, i.e. no communication is in transit.

Similarly to other schemes, TICK augments the local operating system of the nodes to provide new mechanisms to transparently checkpoint the process. Every time a checkpoint is taken by a kernel thread, the following information is saved: register file content, process descriptor information, signal information and signal handlers defined by the process, file descriptors and memory region descriptors.

Performance tests show that for a checkpoint period of one minute, the checkpoint overhead is at most 4% when the checkpoint is stored in memory, and at most 6% when the checkpoint is stored in local disk.

2.6.4 Checkpointing for GPUs

Conventional checkpoint/restart approaches fail to work with applications running with GPUs because those approaches are designed to save only the main memory and CPU state. Therefore, approaches targeting this architecture are needed; next, three approaches for appli-

cation and system level checkpointing are described.

CheCUDA [72] is a user-level checkpointing approach for CUDA applications running on systems with GPUs. This approach relies on the Berkeley Lab Checkpoint/Restart (BLCR) [73] tool to checkpoint the application, which requires that no CUDA objects exist when the checkpoint is taken. The steps taken by CheCUDA are: (i) copy the user data in the GPU to the host memory and (ii) delete the CUDA objects, (iii) copy the application status data to the checkpoint file, (iv) create and initialize CUDA objects, and (iv) copy the user data back to the GPU memory. To collect application status data, CheCUDA includes wrappers for standard CUDA functions, such as `cuMemAlloc()` and `cuMemFree()`, which save information in special resources and object lists. Checkpointing an application with CheCUDA can be done by (i) calling to the `ckptSel()` function, or (ii) using user level signals. Experimental results shows that the overhead imposed by the wrapper functions that collect object information can be up to 80% compared to the regular CUDA functions, and the checkpoint overhead time imposed by CheCUDA in some cases is about the same as the kernel execution time.

CheCL [74] is a user-level checkpointing tool for OpenCL applications. This tool saves only OpenCL objects and hence it works with a conventional system-level checkpointing tool, such as BLCR [73]. Checkpointing is achieved by (i) decoupling the application process from the OpenCL implementation, and (ii) providing a library that contains wrappers for OpenCL objects and functions and records all the necessary information for restoring the GPU state. The CheCL library replaces the `libOpenCL.so` library file, and when it is loaded by an OpenCL application, an additional process is created, i.e. proxy process, that receives all the API calls and then calls the actual API functions. Two checkpoint modes are supported by CheCL: delayed checkpointing mode and immediately checkpointing mode; the former waits for the next synchronization call to checkpoint the data, while in the later a synchronization call is forced and the checkpoint is performed immediately. Experimental results show that the time overhead imposed only by the CheCL library ranges from 10% up to nearly 80%, on average 20%, depending on the number of API calls. Checkpoint and restart time (in seconds) are reported, however, it does not provide much information as no execution time is provided.

Laosooksathit et al. [75] propose a checkpoint mechanism that achieves low checkpoint

overhead by combining CUDA streams and a tool that provides transparent checkpoint/restart to virtual machines. CUDA streams allows to divide and overlap GPU computation and data transfer to the host memory, which reduces the communication overhead. The checkpoint is performed in two steps: (i) the user data is copied back to the host memory, and (ii) the virtual machine tool checkpoints the application state. This mechanism assumes that each kernel synchronization point is a potential checkpoint request, hence, to decrease the number of checkpoints taken two metrics are introduced: the expected cost of performing the checkpoint and the expected cost of skipping the checkpoint.

CHAPTER 3. Unstructured Grid Applications on Graphics Processing Units

3.1 Introduction

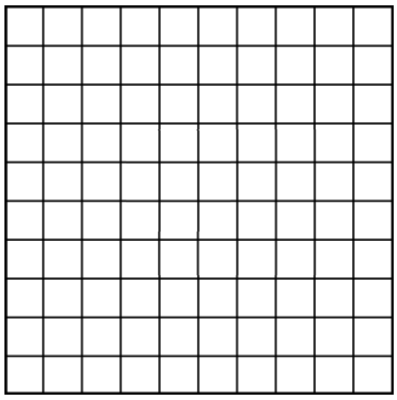
Many fields of scientific research rely on simulations that require the analysis of surfaces. In order to be tractable by computational methods, i.e. to numerically solve partial differential equations, surfaces are discretized into small cells that form a mesh or grid. The grid representation has an impact on the rate of convergence, the solution accuracy and the computation time required. Depending on the complexity of the problem to represent and solve, a structured or unstructured grid is utilized.

3.2 Structured & Unstructured Grids

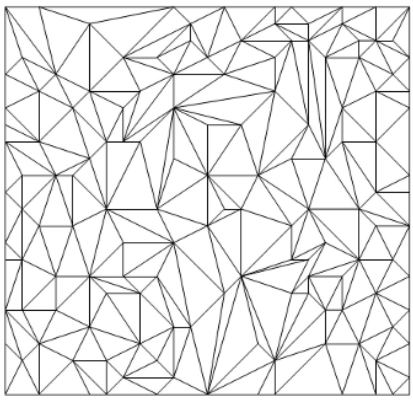
In a structured grid all its interior cell vertices belong to the same number of cells, whereas in a unstructured grid every cell vertex is allowed to belong to different number of cells, see Figure 3.1.

Algorithms are more efficiently implemented in structured grids, and data structures to handle the grid are easy to implement; however, structured grids present poor accuracy if the problem to be solved has curved internal or external boundaries. On the other hand, unstructured grids present more flexibility and higher accuracy to represent problems that have curved boundaries; however, the data structures to handle it are not easy to implement, and also explicit neighboring information should be stored [76]. In general unstructured grids are more utilized because of their flexibility and higher accuracy.

This document focuses on unstructured grids, therefore, the terms unstructured grid and grid are used interchangeably in the next sections.



(a) A structured grid representation



(b) An unstructured grid representation

Figure 3.1: Grid representation of a surface

Every cell in a grid is defined by its shape, i.e. number of sides or faces (N_{faces}), the number of solution points in every face (N_{SPpF}), and the number of solution points inside the cell (N_{inner}). The total number of solution points per cell (N_{SP}) is given by:

$$N_{SP} = N_{faces} \cdot N_{SPpF} + N_{inner} \tag{3.1}$$

Figure 3.2 shows a triangular cell with three solution points per face ($N_{SPpF} = 3$) and one inner solution point ($N_{inner} = 1$).

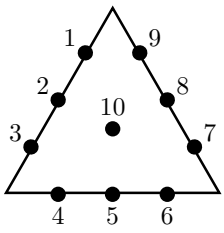


Figure 3.2: A triangular cell with ten solution points

Each solution point has a set of values (N_{par}) that represent the local magnitudes for the set of variables to analyze or parameters at that point, i.e. $v_{i,j}$ with $1 \leq i \leq N_{SP}$ and $1 \leq j \leq N_{par}$. Depending on the research field the parameters may represent pressure, viscosity, velocity, etc.

3.3 Unstructured Grid-based Analysis

Analysis using an unstructured grid is implemented as an iterative method where the values of the variables at each solution point are updated until converges to the solution or reaches a number of iterations. The operations that are carried out in every iteration can be divided into four parts:

- *Local cell analysis*: obtains a coefficient for each solution point based only on the interaction with the other solution points in the same cell.
- *Neighbor cell analysis*: computes a coefficient for each solution point based on the interaction with its neighbor solution point.
- *Update local variables*: the local value of each solution point is updated using a linear combination of the two previously computed coefficients.
- *Update the boundary cells*: the local value of each solution point at the grid boundaries is updated based on the interaction with the other solution points in the same cell.

Next, Algorithm 1 presents the main algorithm for analysis based on unstructured grids.

It is important to notice that in an actual implementation of Algorithm 1, step 13 can be performed in either the step 12 or 18. The selection for either of those depends on the execution type, i.e sequential or parallel, and, if used, the type of parallelism exploited.

As it can be seen in Algorithm 1, the four main stages perform computations based on information stored in main memory, such as the solution point variables, geometry information, and a set of parameters for cell-oriented or neighbor-oriented (edge-oriented) analysis. What is interesting to notice is that although solution point variables and parameters are heavily used in all four main stages, they are accessed with different patterns at every stage. These memory patterns limit data locality between and inside the stages, diminishing efficiency of data caches for reducing memory latency. Therefore, performance of the grid analysis algorithm is limited by memory latency.

Algorithm 1 Analysis using an unstructured grid

```
1: counter ← 0
2: repeat
3:    {Cell-oriented Analysis}
4:    for all cells in grid do
5:       for all solutionPoints in currentCell do
6:          Compute local coefficient based on information of solution points within the same
            cell
7:       end for
8:    end for
9:    {Neighbor or Edge-oriented Analysis}
10:   for all edges in grid do
11:      for all solutionPoints in currentEdge do
12:         Compute local coefficient based on information of the neighbor solution point
13:         Obtain a single local coefficient by performing a linear combination of the two pre-
            viously computed local coefficients
14:      end for
15:   end for
16:   {Updates variables of solution points}
17:   for all solutionPoints in grid do
18:      Update local variables utilizing the local coefficient computed in the previous steps
19:   end for
20:   {Updates boundary condition variables}
21:   for all solutionPoints in gridBoundaries do
22:      Update boundary condition variables based on information of solution points within
         the cell
23:   end for
24:   Check for convergence
25:   counter ← counter + 1
26: until ( Converges ) or ( counter = Max )
```

3.4 Memory Access Pattern

In Section 3.3, the main algorithm for analysis using unstructured grids was introduced, which because of the data access patterns of its main stages is limited by memory latency. In this section we describe the access pattern for the cell-oriented and neighbor-oriented analysis stages in Algorithm 1 because those two stages account for almost 90% of the computation time of the algorithm.

3.4.1 Cell-Oriented Analysis

In cell-oriented analysis, a set of coefficients for each solution point is computed based on its own information as well as the information of the solution points that belong to the same cell. As shown in Figure 3.3 accessing the solution point information is performed in two steps: the first step involves retrieving the pointer to the beginning of the cell in the array of solution point variables, and the second step involves accessing sequentially all the information in the current cell.

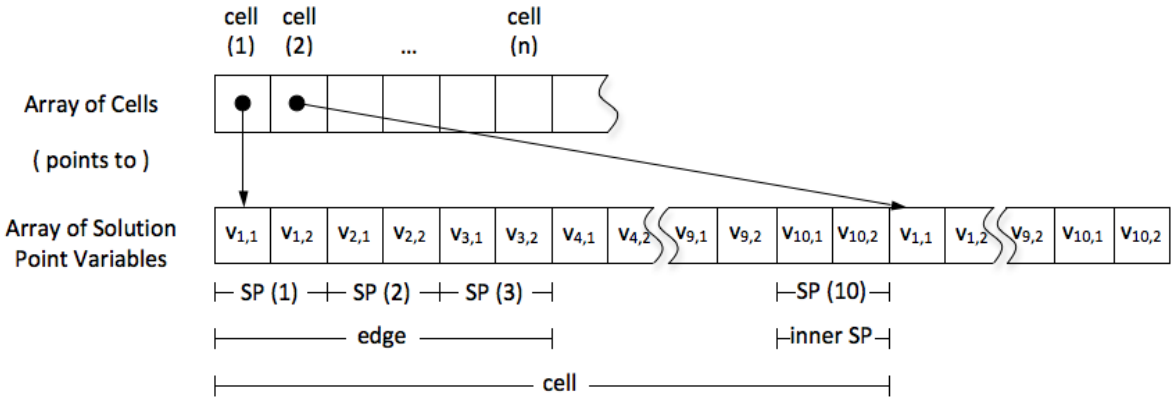


Figure 3.3: Cell-Oriented analysis memory access pattern for a grid with 3-sided cells, three solution points per face, one inner solution point and two parameters per solution point

Because all the cells store the same amount of information, the array with pointers to the beginning of the cell, i.e the array of cells in Figure 3.3, it is not needed. Instead the beginning of a cell in the array solution point variables can be computed as:

$$Ptr = N_{SP} \cdot N_{par} \cdot (\#Cell - 1)$$

As it can be noticed, the solution point variables in every cell are read once and utilized several times, i.e N_{SP} times. Clearly on a uni-threaded solution, cache memories are useful to exploit temporal and spatial locality. Unfortunately, depending on the design of a multi-threaded solution cache memories can exploit only spatial locality, because it is likely that information in cache is replaced as required by different threads. Therefore, the expected performance gain by a multi-threaded solution could be drastically reduced.

3.4.2 Neighbor-Oriented Analysis

In neighbor-oriented (or edge-oriented) analysis, a set of coefficients for each solution point is computed based on its own information and the information of its neighbor solution point, see Figure 3.4.

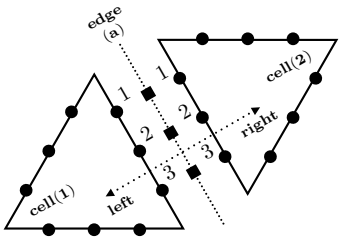


Figure 3.4: Cell iterations between neighbors

Unlike cell-oriented analysis that traverses the grid at cell-level, edge-oriented analysis traverses the grid at edge-level. Figure 3.5 shows that accessing the solution point information is done in three steps: the first step involves retrieving the pointer to the solution point, in the second step the pointer to the left and right solution point variables are retrieved, and the third step involves accessing the two solution points variables.

Alike cell-oriented analysis, the first step is trivial as the pointers to the solution points can be easily computed as $2 \cdot (\#edge - 1) \cdot N_{SPpF}$.

Unlike cell-oriented analysis, left and right solution point variables are not physically adjacent, and information is read and used only once, hence, either on a uni-threaded or multi-threaded solution the cache memories do not help to reduce memory latency.

In the last stage the solution point variables are updated utilizing only current solution point information and coefficients, i.e. read and utilized once. Since coefficients and solution

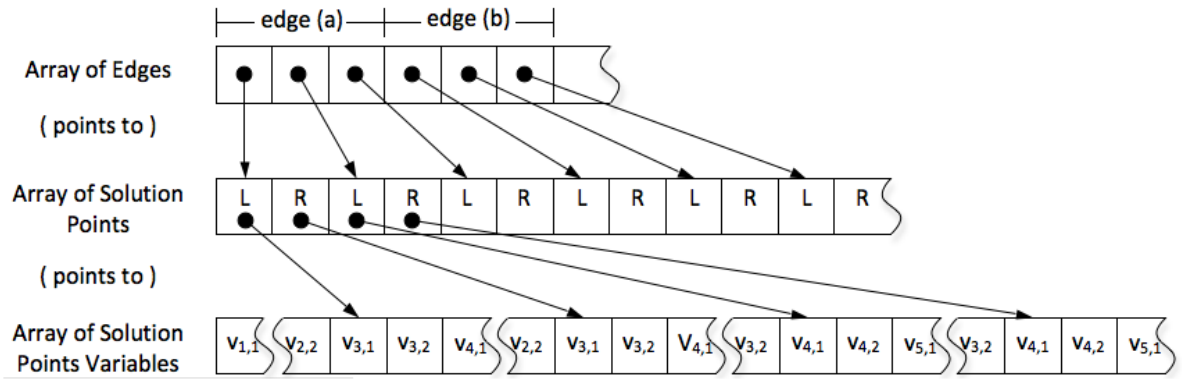


Figure 3.5: Edge-Oriented analysis memory access pattern for a grid with 3-sided cells, three solution points per face, one inner solution point and two parameters per solution point

point variables arrays are processed sequentially, cache memories can take advantage of spatial locality, and by this way help to reduce memory latency for both uni-threaded and multi-threaded solutions.

3.5 Performance Considerations

In the CUDA platform, for execution purposes thread blocks are assigned to Streaming Multiprocessors (SMs) whereas for scheduling purposes the threads belonging to the same thread block are grouped into *warps*. Instructions of a warp are executed one at a time on all its threads.

GPUs achieve high performance by hiding memory access latency, which is possible by switching warp execution between warps that are waiting for long latency operations to finish and warps that are ready to continue execution. Under this premise performance on GPUs is mainly dependent on SM occupancy and global memory access.

3.5.1 Streaming Multiprocessor Occupancy

In a GPU a large number of warps active in a SM, i.e high occupancy, is needed to tolerate long latency operations. The maximum occupancy in terms of number of threads, blocks and warps that can be achieved is determined by hardware specifications [11]. However, achieving maximum occupancy depends on the number of registers and amount of shared memory utilized by each thread block.

The number of registers utilized by the threads in the active thread blocks cannot be greater than the maximum number of registers of the SM. In the same way, the amount of shared memory utilized by the active thread blocks cannot be greater than the total amount of shared memory of the SM. Hence, the number of active thread blocks can be computed as follows.

$$ThreadBlocks = \min\left(\left\lfloor \frac{TotalSharedMemory}{SharedMemoryPerBlock} \right\rfloor, \left\lfloor \frac{TotalRegisters}{RegistersPerBlock} \right\rfloor\right) \tag{3.2}$$

Now, taking into consideration the maximum number of active thread blocks imposed by hardware specifications, the number of active threads is given by:

$$\begin{aligned} ActiveWarps = \min\Big(&maxActiveWarps, maxTBlocksPerSM \times WpTB, \\ &ThreadBlocks \times WpTB\Big) \end{aligned} \tag{3.3}$$

In the previous equation $WpTB$ refers to the number of *warps per thread block*, which depends on the number of threads per block as shown next.

$$WpTB = \left\lceil \frac{ThreadsPerBlock}{ThreadsPerWarp} \right\rceil \tag{3.4}$$

Equation 3.3 defines the number of warps that can be active on a SM and it shows that the highest value is limited by register or shared memory usage, by the maximum number of active thread blocks and active warps specified by the hardware implementation.

Table 3.1 describes the parameters utilized in Equations 3.2, 3.3 and 3.4 that influence the occupancy. The first five parameters in the table are architecture dependent, whereas the last two parameters are application dependent.

3.5.2 Global Memory Access

A GPU implements different types of memory for storing data: global memory, constant memory, texture memory, shared memory and registers. This memory structure allows to reduce global memory accesses and collaboration among threads in the same thread block. In terms of latency, global memory access is the slowest whereas registers are the fastest.

Since the GPU execution model requires that the information is first placed in global memory and then accessed by the GPU application, it is necessary to optimize global memory

Table 3.1: Parameters that influence SM occupancy

Parameter	**Description**
TotalSharedMemory	Total amount of shared memory per SM
TotalRegisters	Number of registers available per SM
ThreadsPerWarp	Number of threads that are grouped into warps
maxActiveWarps	Maximum number of warps that can be active in a SM
maxTBlocksPerSM	Maximum number of Thread Blocks that can be active in a SM
SharedMemoryPerBlock	Amount of shared memory utilized by a thread block
RegistersPerBlock	Number of registers utilized by a thread block

access. Global memory access can be optimized by achieving peak bandwidth and by reducing the number of accesses.

Although GPU provides large bandwidth for global memory operations, the access pattern of the threads of a warp can reduce considerably the achieved bandwidth. To achieve peak bandwidth usage, the GPU coalesces warp memory operations into two or four memory transactions depending on the size of the words accessed. Therefore, warp memory access should be organized in such a way that threads access adjacent memory locations. Depending on the memory access pattern the number of memory transactions per warp is limited as follows.

$$\begin{aligned} 2 \leq MemTransactions &\leq MaxMemTransactions \\ &= ThreadsPerWarp \end{aligned} \tag{3.5}$$

When data is reutilized it is possible to reduce the number of global memory accesses by storing the data either in registers or in shared memory. Shared memory is common for all the threads in the thread block, which allows collaboration among them. Since shared memory is organized in banks, to avoid bank conflicts threads should access data in different banks.

3.6 Implementation of Unstructured Grid Applications on GPUs

This section presents the implementation and performance analysis of Algorithm 1 introduced in Section 3.3. Since the performance analysis require hardware-dependent parameters, the NVidia Tesla T10 GPU is used on the remainder of this chapter. The technical specifications of the Tesla GPU are shown in Table 3.2.

Table 3.2: NVidia Tesla T10 Technical Specifications

Parameter	**Value**
Number of Streaming Multiprocessors	30
Number of Streaming Processors per SM	8
Number of 32-bit Registers per SM (TotalRegisters)	16 K
Shared Memory per SM (TotalSharedMemory)	16 KB
Warp Size (ThreadsPerWarp)	32
Active Warps per SM (maxActiveWarps)	32
Active Thread Blocks per SM (maxTBlocksPerSM)	8

3.6.1 Streaming Multiprocessor Occupancy

Due to space constraints this section presents the implementation and analysis only for the cell-oriented stage, analysis for the edge-oriented stage is similar.

As mentioned in Section 3.3, in cell-oriented analysis every solution point computes a coefficient based on its own variables as well as the variables of the solution points in the same cell, hence, there is no collaboration between cells.

The straight forward implementation maps one cell to one thread block, where each solution point is represented by one thread. In this implementation the number of thread blocks is equal to the number of cells and the number of threads per thread block is equal to the number of solution points.

Using Equation 3.2 and the fact that acording to Table 3.2 the maximum number of active thread blocks per SM is eight:

$$\begin{aligned} 8 &= \left\lfloor \frac{16 \text{ KB}}{SharedMemoryPerBlock} \right\rfloor \\ &\Longrightarrow SharedMemoryPerBlock = 2 \text{ KB} \\ 8 &= \left\lfloor \frac{16 \text{ K}}{RegistersPerBlock} \right\rfloor \\ &\Longrightarrow RegistersPerBlock = 2 \text{ K} \end{aligned} \tag{3.6}$$

Now, assuming triangular cells and using Equation 3.1 it is possible to approximate:

$$\begin{aligned} SharedMemoryPerThread &= \frac{2 \text{ KB}}{N_{SP}} \\ &\approx \frac{2 \text{ KB}}{3 \times N_{SPpF}} \\ RegistersPerThread &= \frac{2 \text{ K}}{N_{SP}} \\ &\approx \frac{2 \text{ K}}{3 \times N_{SPpF}} \end{aligned} \tag{3.7}$$

For a large value of solutions points per face, i.e. $N_{SPpF} = 9$, it is possible to have approximately up to 90 32-bit words per thread (or solution point) for calculations between shared memory and registers, which it is enough. Therefore, it is possible to have eight active thread blocks per SM because shared memory and registers do not impose limitations on the number of active thread blocks per SM.

Finally, from Equation 3.4 the number of warps per thread block:

$$\begin{aligned} WpTB &= \frac{N_{SP}}{ThreadsPerWarp} \\ &= \frac{3 \times 9 + N_{inner}}{32} \\ &\approx 1 \end{aligned} \tag{3.8}$$

Therefore, there are eight active warps per SM, which means that only 25% of the occupancy is achieved by this implementation.

Clearly, it is necessary to increase occupancy to fully hide global memory access and to keep the streaming processors working. To increase occupancy it is necessary to increase the number of threads per thread block, i.e. to increase the number of cells per thread block. Assuming that the number of active thread blocks is not limited by register usage, the number of active thread blocks is given in Equation 3.9.

$$\begin{aligned} ThreadBlocks &= \left\lfloor \frac{16 \text{ KB}}{SharedMemoryPerBlock} \right\rfloor \\ &= \left\lfloor \frac{16 \text{ KB}}{\text{S. Mem per Cell} \times \text{Cells per TBlock}} \right\rfloor \\ &\leq 8 \end{aligned} \tag{3.9}$$

In Equation 3.9 the number of *cells per thread block* is chosen such that the number of *active thread blocks per SM* does not exceed eight, therefore, it is not limited by either shared

memory or hardware specifications. Now, the number of active warps per SM can be computed as follows.

$$\begin{aligned} ActiveWarps &= ThreadBlocks \times WpTB \\ &= \left\lfloor \frac{\text{16 KB}}{\text{S. Mem per Cell} \times \text{Cells per T. Block}} \right\rfloor \times \left\lceil \frac{N_{SP} \times \text{Cells per T. Block}}{32} \right\rceil \\ &\leq 32 \end{aligned} \tag{3.10}$$

From Equation 3.10 a good approximation for testing purposes can be derived:

$$\begin{aligned} ActiveWarps &= \frac{512 \times N_{SP}}{\text{S. Mem per Cell}} \\ &= \frac{512}{\text{Shared Mem per Solution Point}} \end{aligned} \tag{3.11}$$

A similar analysis for the used registers provides the following approximation:

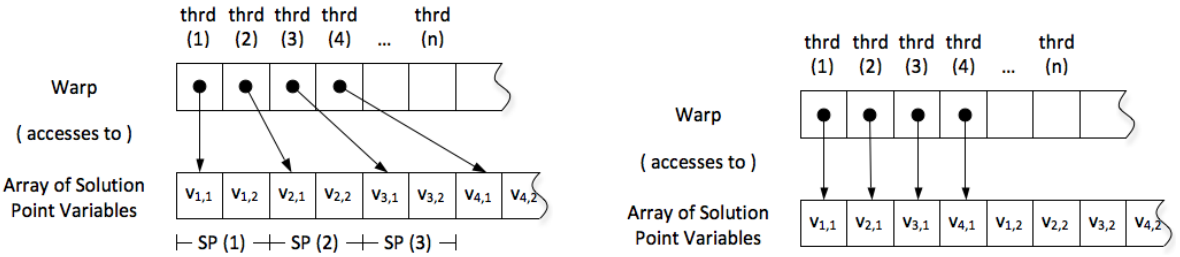
$$ActiveWarps = \frac{500}{\text{Registers per Solution Point}} \tag{3.12}$$

3.6.2 Global Memory Access

Section 3.4 presented the memory access pattern for the cell-oriented analysis. In the GPU implementation every thread accesses the information of a single solution point. According to the algorithm presented in Section 3.3, every thread accesses non-contiguous memory locations, which increases the number of memory transactions non-linearly depending on the number of parameters stored at each solution point. Figure 3.6a shows non-contiguous memory access for a case with two parameters.

In order to access contiguous memory locations, the data have to be reorganized as shown in Figure 3.6b. The number of memory transactions for the reorganized data implementation can be computed using Equation 3.13

$$\begin{aligned} \#\text{ Transactions} &= 2 \times \left\lceil N_{par} \times \frac{1/2 \times ThreadsPerWarp \times sizeof(float)}{\text{Transaction Size} = 64 \text{ bytes}} \right\rceil \\ &= 2 \times N_{par} \end{aligned} \tag{3.13}$$



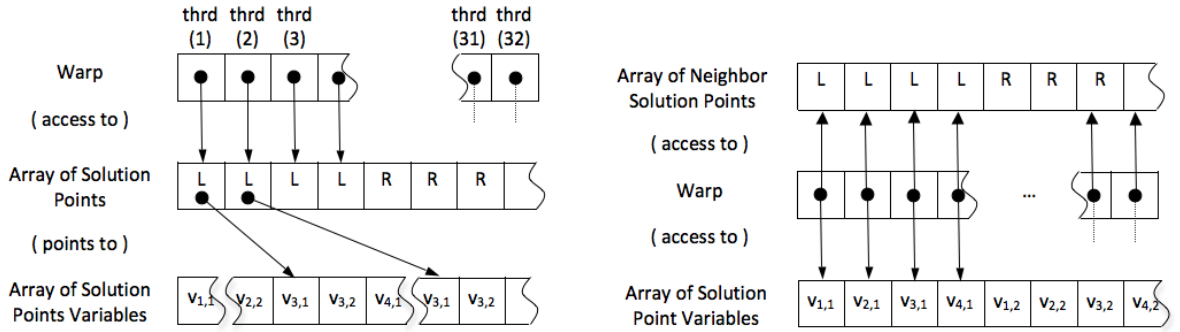
(a) Before data reorganization

(b) After data reorganization

Figure 3.6: Thread memory access on the cell-oriented stage

As it can be seen in Equation 3.13, for single precision parameters the number of memory transactions for a warp increases linearly with the number of parameters.

Memory access pattern for edge-oriented analysis, is performed in two stages: first the data geometry is read and then the solution point information. As shown in Figure 3.7a, geometry information is accessed sequentially, therefore, coalescing memory transfers is possible. However, due to the memory access pattern, solution point information is accessed in a random-like fashion, making not possible to coalesce memory transfers, which leads to potentially one memory transaction per thread.



(a) Before data reorganization

(b) After data reorganization

Figure 3.7: Thread memory access on the edge-oriented stage

Under this considerations, the total number of memory transactions is approximated as:

$$\#\ \text{Transactions} = 2 \times \#\text{Edges} \times N_{SPpF} \times N_{par} \tag{3.14}$$

Intuitively equation 3.14 represents all the non-coalesced memory accesses. Since the num-

ber of memory transactions increase linearly with the total number of threads, the objective is to decrease the number of memory access or in other words avoid the random-like memory access.

The strategy used to reduce the number of memory transactions is depicted in Figure 3.7b. The main difference is that instead of traversing the grid through the edges, the grid is traversed through cells, therefore, the memory pattern for the solution point variables changes such that the threads read contiguous memory locations. The neighbor pointer for all the solution points is stored in a new structure, and it is accessed sequentially by the threads. However, the neighbor information is still accessed in a semi random-like fashion. In the same way as with the previous scheme the total number of memory transactions can be approximated as:

$$\#\text{ Transactions} = N_{par} \times \left[N_{Faces} \times \#\text{Cells} \times N_{SPpF} + 2 \times \frac{\#\text{Cells}}{\text{Cells per T. Block}} \times WpTB \right] \tag{3.15}$$

The terms inside the brackets in Equation 3.15 represent the neighbor information and the solution point accesses, respectively.

In general the number of memory transactions defined by Equation 3.14 is bigger that the one defined by Equation 3.15.

This change in the memory access pattern changes the algorithm described in Section 3.3, which is described in Algorithm 2.

It is important to notice that the second approach generates $N_{faces} \times \#Cells \times N_{SPpF}$ threads, which is bigger compared to the number of threads generated by the first approach $\#Edges \times N_{SPpF}$, however, this does not affect negatively the performance because this algorithm is memory latency limited.

3.7 Implementation Results

Computational Fluid Dynamics (CFD) is a scientific area that analyze and solve problems involving fluid flows utilizing numerical approaches. Aerospace engineering is one of the fields that led the CFD development. The application utilized in this section solves the Navier-Stokes equations on unstructured grids utilizing a high order correction procedure via reconstruction

Algorithm 2 Unstructured grid analysis on GPU

```
1: counter ← 0
2: Initialize initial state
3: repeat
4:    {Cell and Neighbor Analysis}
5:    for all cells in grid do
6:       for all solutionPoints in currentCell do
7:          Compute local coefficient based on information of the neighbor solution point
8:          Compute local coefficient based on information of solution points within the same
            cell
9:          Obtain a single local coefficient by performing a linear combination of the two coef-
            ficients previously computed
10:      end for
11:   end for
12:   {Updates variables of solution points}
13:   for all solutionPoints in grid do
14:      Update local variables utilizing the local coefficients computed in the previous steps
15:   end for
16:   {Updates boundary condition variables}
17:   for all solutionPoints in gridBoundaries do
18:      Update boundary condition variables based on information of solution points within
         the cell
19:   end for
20:   Check for convergence
21:   counter ← counter + 1
22: until ( Converges ) or ( counter = Max )
```

methods. Details on this method are omitted, for the further details refer to the paper by Wang [77].

The CFD application was originally implemented and optimized for running on CPUs. The most important optimization techniques utilized in the CPU implementation are loop unrolling, improved memory access and cache utilization.

In this section we present the results of the implementation of the CFD application on a system composed by a Intel Xeon 3GHz quad-core processor and a GPU NVidia Tesla T10 GPU utilizing the algorithm proposed in this document. For comparison purposes we show the speedup achieved by the GPU implementation without occupancy optimization (GPU_1) and the one with occupancy optimization (GPU_2). The speedup is computed considering the original CPU implementation.

Table 3.3: GPU Implementation of a CFD Application

N_{SP}	No Occupancy Optimization			Occupancy Optimization			Speedup
	Warps per SM	Threads per Warp	Speedup (CPU/GPU_1)	Warps per SM	Threads per Warp	Speedup (CPU/GPU_2)	Speedup (GPU_1/GPU_2)
3	8	3	20	8	32	27	1.35
6	8	6	31	8	32	45	1.45
10	8	10	41	8	32	63	1.54
15	6	15	37	8	32	82	2.22
21	3	21	43	8	32	88	2.05

As it can be seen on Table 3.3, by improving the occupancy, the speed is improved by at least 35% and in some cases the speedup achieved is doubled. The speedup of the occupancy optimized implementation increases linearly if compared to the non-optimized implementation, which is due to the higher occupancy achieved. In is interesting to notice that for cells with 15 solution points, the speedup is the highest, which is due to: (i) the occupancy achieves almost full occupancy (255 threads), only one thread is wasted, and (ii) a cell almost corresponds to a half warp, hence, frequently one memory transaction reads information for one cell.

3.8 Conclusions

Implementation on a NVidia Tesla GPU of the ustructured grid-based analysis algorithm was analyzed in terms of hardware occupancy and global memory access. This analysis led us to propose an algorithm that achieves higher occupancy and more efficient global memory access than the original algorithm. The actual GPU implementation achieved a speedup of more than 80 times compared to the CPU version.

The edge-oriented analysis was shown to be troublesome because of the random-like memory access, which linearly increased the number of memory transactions. In order to reduce the number of memory transactions, the edge-oriented analysis was transformed into a cell-oriented analysis. This new approach reduces the number of memory transactions, but at the same time increases the number of threads generated. However, this was not an issue because unstructured grid applications are memory latency limited, which means that computation is overlapped with memory operations.

CHAPTER 4. Techniques for the Parallelization of Unstructured Grid Applications on Multi-GPU Systems

4.1 Introduction

As the popularity of GPUs for accelerating scientific applications grows, the size and complexity of scientific applications grow as well. GPUs are good for processing big amounts of data, however, the computational model of GPUs requires all the information to be stored in its global memory [18]. This requirement limits the memory footprint of programs to the amount of global memory. Currently GPUs are built with up to 6GB of global memory, which limits the size of tractable problems. To overcome this limitation the computational problem needs to be partitioned in smaller pieces and executed in all the available hardware in the system.

Multi-GPU systems are becoming popular as they allow to exploit higher levels of parallelism and also as an effort to make memory-limited problems tractable by GPUs. In multi-GPU systems, the processing and data are partitioned into the GPUs available, however, parallelizing applications is not trivial due to the data dependencies and the inherent complexity of the application. Moreover, parallelized applications impose data transfer through the PCIe interconnection network, which is a slow operation that reduces the overall application performance.

This chapter presents the analysis and parallel implementation of unstructured grid applications on multi-GPU systems.

4.2 Data Dependencies Analysis

To determine the implementation details it is needed to understand the data flow and, therefore, the data dependencies among the main stages of the algorithm. Depending on the data dependencies, parallel execution can be exploited.

Next, the data flow of the unstructured grid analysis algorithm (Algorithm 2 presented in Chapter 3) is introduced and the data dependencies are represented using a Dependence Graph.

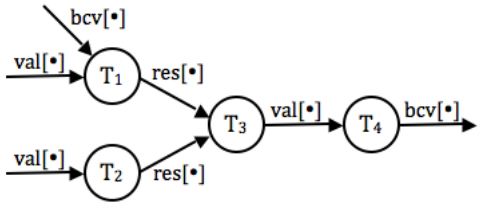
Figure 4.1a shows the stages in the main loop of the algorithm: neighbor analysis (T_1), cell analysis (T_2), local variables update (T_3) and boundary condition variables update (T_4), as well as the data flow among them. For simplicity only the data that is shared and modified by the different stages is shown; for instance the values of the variables at each solution point, represented as val, is an input for T_1 and T_2 and an output for T_3.

Table 4.1 presents the description and size of the variables used in Figure 4.1a. In the column *Size*, $\#Cells$ refers to the total number of cells of the grid, $\#BoundaryFaces$ is the number of faces at the boundaries of the grid, N_{SP}, N_{par} and N_{SPpF} were introduced in Section 3.2.

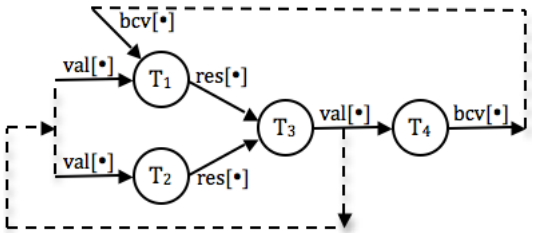
Table 4.1: Description of the variables utilized in the data flow analysis of Algorithm 2

Name	**Description**	**Size**
val	Variables for each Solution Point	$\#Cells \times N_{par} \times N_{SP}$
bcv	Variables for S.P. at the grid boundaries	$\#BoundaryFaces \times N_{par} \times N_{SPpF}$
res	Coefficients for each Solution Point	$\#Cells \times N_{par} \times N_{SP}$

Since the main algorithm is iterative, to fully understand the data dependencies among the main stages of the algorithm, it is necessary to analyze the data flow in two consecutive iterations. Figure 4.1b shows the data flow considering two iterations i and $i+1$, represented by solid and dotted lines, respectively.



(a) Main stages of the algorithm and data flow



(b) Data flow for iteration i (solid line) and $i+1$ (dotted line) of the main loop

Figure 4.1: Data flow of the algorithm for analysis based on unstructured grids

Based on Figure 4.1b it is possible to define the dependence graph for the algorithm. The

dependence graph represents the stages of the algorithm as nodes and the data dependencies as directed arrows. A data dependency is represented by $T_i \rightarrow T_j$, which denotes that task T_j is dependent on data produced by task T_i [78]. Tasks that do not have data dependencies among each other can take advantage of task parallelism, whereas dependent tasks might take advantage of data parallelism.

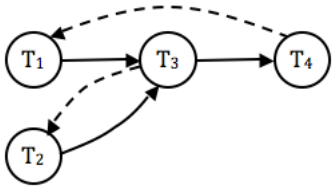


Figure 4.2: Data dependencies graph for the algorithm for analysis based on unstructured grids

The dependence graph presented in Figure 4.2 shows the dependencies within one loop iteration as well as the dependencies produced by the next consecutive iteration, represented by solid and dotted lines, respectively. Dependencies within one iteration imposes limitations on the parallelism achieved, i.e. data parallelism vs. task parallelism. Dependencies among two different iterations defines the need of synchronization points, in case of parallel execution.

In computing systems where all the computation is done by a single computing device, i.e. CPU or GPU, the stages described in Algorithm 2 are executed sequentially. Figure 4.3 presents the sequential execution of one iteration of the main loop in Algorithm 2. Notice that the arrow between T_1 and T_2 does not represent a data dependency, but instead it is used to indicate sequential execution.

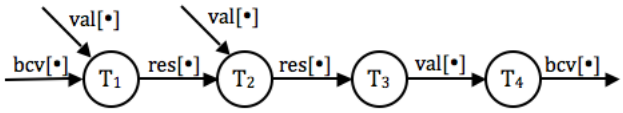


Figure 4.3: Sequential execution of the four stages for analysis based on unstructured grids

4.3 Single GPU Analysis

Since the CPU and GPU have separate memory spaces, the data used by a kernel should be transferred to the GPU memory before any computation can take place. Hence, the memory of the GPU limits the memory footprint of a application executed in a GPU.

When the memory footprint of an unstructured grid-based application does not fit in the global memory of the GPU, the computation should be decomposed into smaller pieces and executed sequentially in the GPU. In this scenario, the data might be decomposed as well and loaded into GPU memory when it is needed. Figure 4.4 presents the single-GPU execution when only either half of the geometry information ($geom_x[\cdot]$) or half of the initial values ($init_x[\cdot]$) and half of the parameter information ($par_x[\cdot]$) fits in GPU memory. Therefore, when the first or second half of the information is needed by the computation, it is loaded/unloaded from memory.

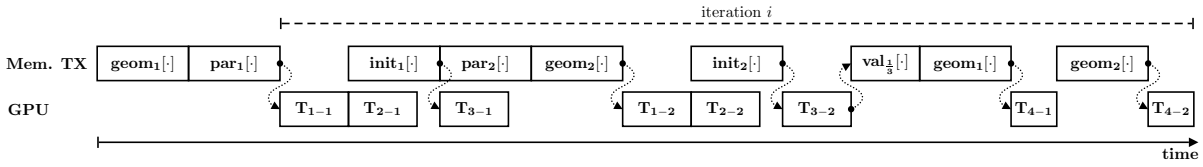


Figure 4.4: Single-GPU execution of the unstructured grid application when the memory footprint does not fit GPU memory

The case depicted in Figure 4.4 presents a saving of approximately 48% of GPU memory and imposes a communication overhead that roughly can be approximated as: $\frac{3}{2} \cdot geom[\cdot] + \frac{1}{2} \cdot par[\cdot] + \frac{1}{3} \cdot init[\cdot]$.

4.4 Parallelism Analysis

An algorithm subject to parallelization can take advantage of Data Parallelism and/or Task Parallelism [79]. In data parallelism, task T_k is divided in N partitions and executed in parallel, where every task partition modifies different pieces of data. In general the number of partitions N is less than or equal to the number of compute devices available, M. In task parallelism, tasks T_k and T_l are executed in parallel if there are not data dependencies between them.

Although parallel execution presents several advantages, it introduces additional complexity to the algorithm and program code that can result in unexpected program behavior. For instance, race conditions could be introduced if no synchronization points are defined.

Alike other parallel systems, the amount of parallelism achieved by systems with multiple GPUs depends fundamentally on data dependencies. Data dependencies have a dual negative

effect on parallelism: (i) limits the task parallelism achievable, and (ii) imposes data transfers among tasks, which might be translated as data transfers between GPUs. Data transfers between GPUs have an adverse effect in the performance if not carefully considered, because any performance gain achieved with parallelism can be negated by the transfer latency.

The computational model of GPUs requires that the data used in the computations should be stored in the device memory before performing the computations. Therefore, data that is shared among several GPUs should be copied in the device memory of all the GPUs involved in the computation. In addition to that requirement, to transfer data between GPUs, the data first is transfered to the host memory, and then the data is transfered to the device memory of the other GPU.

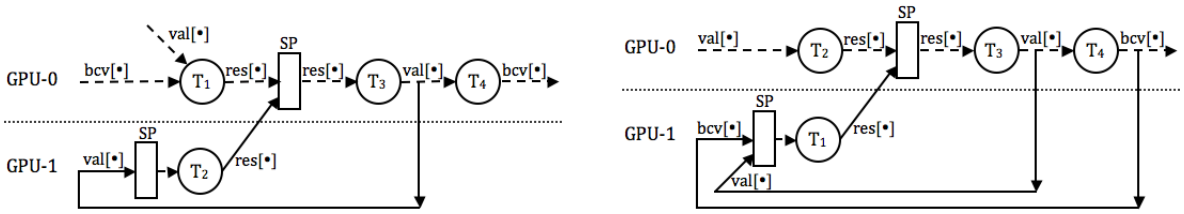
Next, data and task parallelism for unstructured grid applications are analyzed in terms of data dependencies and the size of data transfers required.

4.4.1 Task Parallelism

The task parallelism possible to achieve in Algorithm 2 can be determined from the dependence graph presented in Figure 4.2. Due to data dependencies only tasks T_1 and T_2 can be executed in parallel, resulting in two possible execution escenarions. Figure 4.5 presents the two possible execution scenarios: (i) Tasks T_1, T_3 and T_4 executed in GPU-0, and T_2 executed in GPU-1, and (ii) Tasks T_2, T_3 and T_4 executed in GPU-0, and T_1 executed in GPU-1. Data transfers between GPUs, represented using solid lines in Figure 4.5, are asynchronic, hence, to avoid unexpected behavior, synchronization points (SPs) are included before executing tasks that depend on data transferred from another GPU.

Although these two execution scenarios are very similar, they impose different requirements in terms of data transfers. Next, the size of data transfers between GPUs is analyzed.

As it can be seen in Figure 4.5a, task T_2 generates a coefficient (res) for every solution point based on the information of all the solution points in the same cell, i.e. val. Arrays val and res are transfered from and to, respectively, the device memory of the other GPU, therefore, according to the variable sizes defined in Table 4.1 the number of floating point



(a) GPU-0: T_1, T_3 and T_4 / GPU-1: T_2

(b) GPU-0: T_2, T_3 and T_4 / GPU-1: T_1

Figure 4.5: Task parallelism and data transfers of the algorithm for analysis based on unstructured grids

elements transfered is given by next equation.

$$Size_{TPa} = 2 \times [\ \#Cells \times N_{SP} \times N_{par}\ ] \tag{4.1}$$

Using a similar argument, Figure 4.5b shows that task T_1 receives as inputs the values of the solution points for all the cells (val) and the boundary conditions values (bcv), and generates a coefficient (res) for every solution. Since T_1 is executed in a separate GPU, these three sets of data are transfered to and from another GPU. The number of floating point elements transfered is given by:

$$\begin{aligned} Size_{TPb} &= 2 \times [\ \#Cells \times N_{SP} \times N_{par}\ ] + \#BoundaryFaces \times N_{SPpF} \times N_{par} \\ &= Size_{TPa} + \#BoundaryFaces \times N_{SPpF} \times N_{par} \end{aligned} \tag{4.2}$$

Equation 4.2 shows that the scenario depicted in Figure 4.5a is more efficient than the one depicted in Figure 4.5b. However, in either of the two task parallelism scenarios, GPU-1 remains inactive while GPU-0 is busy performing tasks T_3 and T_4.

4.4.2 Data Parallelism

Task parallelism, introduced in Section 4.4.1, increases the performance by executing in parallel tasks T_1 and T_2. However, data dependencies impose large data transfers and limitations in terms of hardware utilization.

Data parallelism can improve the hardware utilization and might reduce the data transfers by partitioning tasks into N smaller pieces that are executed in parallel on the computing devices available and access only to data stored in local memory. In this context $T_{i\text{-}k}$ denotes

the k_{th} partition of task T_i that is executed on the k_{th} computing device, and $array_k[\cdot]$ denotes the subset of $array[\cdot]$ accessed by task $T_{i\text{-}k}$ and stored in the local memory of the k_{th} computing device. In this work it is assumed that data subsets holds $\bigcup_{i=1}^{N} array_i = array$ and $\bigcap_{i=1}^{N} array_i = \emptyset$.

Ideally, tasks should be divided in such a way that no data transfers among GPUs are needed, i.e. $T_{i\text{-}k}$ accesses only data in $array_k[\cdot]$, however, this is only possible if there are no data dependencies among task partitions.

Data dependencies among task partitions are represented by $T_{i\text{-}k} \rightarrow T_{j\text{-}l}$, which denotes that the l_{th} partition of task T_j depends on data generated by the k_{th} partition of task T_i. The data dependencies that involve tasks executed on different computing devices, i.e $k \neq l$, imposes data communication between those computing devices.

As introduced in Section 4.2, tasks T_2 and T_3 access solution point information, i.e. parameters and coefficients, in sequential order grouped by cells. Hence, the subsets of solution point information read and/or updated by task partitions does not overlap among each other, or in other words there is no data dependencies among task partitions. On the other hand, tasks T_1 and T_4 access solution point information sequentially, grouped by cells, and also semi-randomly, grouped by neighboring relationship. Therefore, the subsets of solution point information accessed by task partitions does overlap among each other, which potentially introduces data dependencies among task partitions.

Figure 4.6 presents the data flow between tasks as solid arrows and data transfers among GPUs as dotted arrows for a parallel execution of Algorithm 2 on two GPUs. For simplicity only data that is shared and modified by different stages is shown.

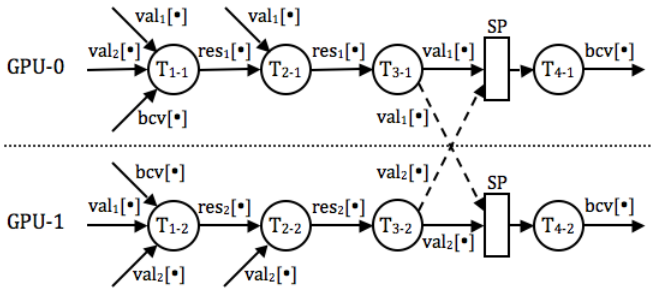


Figure 4.6: Task parallelism and data transfers of the algorithm for analysis based on unstructured grids

As stated previously, tasks T_2 and T_3 have no data dependencies, i.e. $T_{2\text{-}k}$ and $T_{3\text{-}k}$ access only to $res_k[\cdot]$ and $val_k[\cdot]$, whereas tasks T_1 and T_4 have data dependencies that introduce data transfers among GPUs.

As it can be seen in Figure 4.6, $T_{4\text{-}1}$ utilizes the solution point information updated by $T_{3\text{-}2}$, $val_2[\cdot]$, i.e. $T_{3\text{-}2} \rightarrow T_{4\text{-}1}$, therefore, $val_2[\cdot]$ is transferred from GPU-1 to GPU-0. Similarly the data dependence $T_{3\text{-}1} \rightarrow T_{4\text{-}2}$ requires data transfer of $val_1[\cdot]$ from GPU-0 to GPU-1. Since data sets val_k do not overlap among each other, the size of the data transferred is given by Equation 4.3.

$$Size_{DP} = \#Cells \times N_{SP} \times N_{par} \tag{4.3}$$

By comparing Equations 4.1 and 4.3, that corresponds to the task and data parallel execution approaches, it can be clearly seen that data parallel imposes less data transfer overhead.

In terms of computing devices usage, data parallelism presents better utilization since the workload is evenly divided among the computing devices available.

4.5 Overlapping Computation and Communication

In Section 4.4, task and data parallelism were introduced and analyzed for the the algorithm of analysis based on unstructured grids. For data parallelism, a case with two GPUs was considered, and it was clear that it achieves better performance and higher hardware utilization and reduces communication overhead compared to task parallelism.

Although the amount of data communication overhead was reduced using data parallelism, it still reduces the overall performance and it can be critical when the communication time is bigger than the computation time. Figure 4.7 shows the timing for tasks execution and data communication. It can be seen that during the data communication time, t_{TX}, both GPUs remain inactive.

In systems with multiple GPUs, communication between GPUs is done through the PCI-Express (PCIe) interconnection network. The PCIe interconnection network is a point to point interconnection that can have from 1 to 32 lanes per direction for communication, in practice PCIe has a power of two number of lines. In PCIe 2.0 each lane transmits up to 500 MB/sec

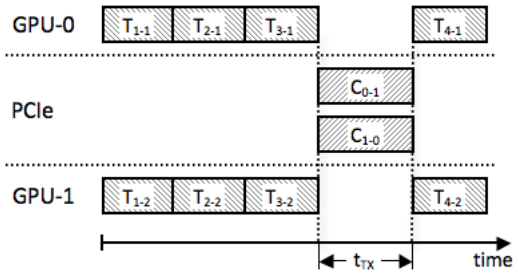


Figure 4.7: Data parallel execution timings

of data; for instance PCIe x16 can deliver up to 16 GB/sec of bandwidth, combining both directions [80]. Data transmission time in the PCIe interconnection network increases linearly with the size of the data transmitted.

By inspecting Equation 4.3 it can be noticed that as the size of the grid, i.e. number of cells, increases, more data is communicated using the PCIe interconnection network, which increases idle times in the GPUs.

Computation-communication overlapping is a technique successfully utilized to minimize the effects of communication overhead [81]. This technique requires to partition both the task that generates the data and the data transfers as well.

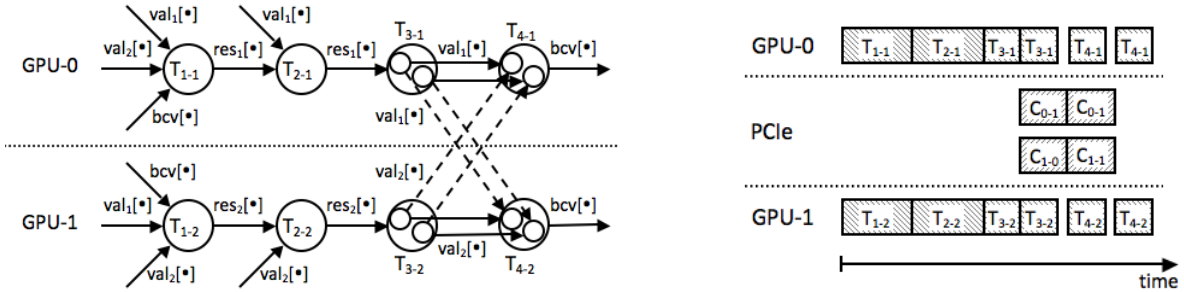
In general, in multiple GPU applications it is possible to overlap computation with communication since data movement between GPUs is explicitly coded in the program.

Depending on the computation and communication time, the latter can be partially or completely minimized. The best scenario is when the computation time is larger than the communication time because the communication can be totally hidden. Figure 4.8a presents the overlapping computation-communication technique applied to the data parallel execution previously presented.

In the case depicted in Figure 4.8b the computation time is shorter than the communication time, hence, the GPUs still remain idle, however, as it can be seen in Equation 4.4 the idle time is shorter compared to the non overlapping computation-communication case presented in Figure 4.7.

$$t_{idle} = t_{TX} - \left[\frac{t_{T3} + t_{T4}}{2}\right] < t_{TX} \tag{4.4}$$

It is important to notice that in the approach proposed in Figure 4.8a the communication



(a) Execution of tasks T_3 and T_4 overlapped with data communication

(b) Timing for the overlapped execution

Figure 4.8: Computation-communication overlapped execution

from GPU-0 to GPU-1 and from GPU-1 to GPU-0 are performed at the same time. Although PCIe implements separate lanes for point to point communication, all the lanes share the switch, which results in additional overhead when more than one device is communicating data. The ideal scenario is when only one GPU communicates data at each time because it reduces the pressure on the PCIe interconnection network, see Figure 4.9.

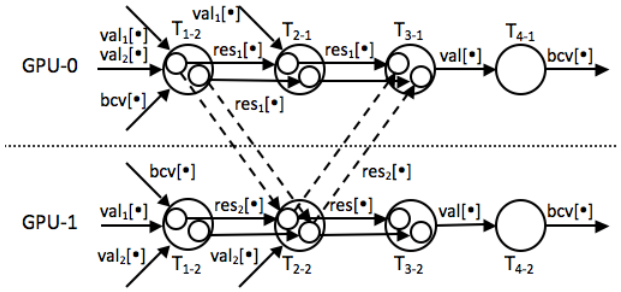


Figure 4.9: A more efficient implementation of computation-communication overlap

The approach presented in Figure 4.9 might require additional space for data transmitted, and also some task partitions might perform extra computation, which is not an issue because the extra computation can be performed while the data transfer is taking place. For instance in this scenario, additional space for the coefficients of the solution points is required, and also partitions $T_{2\text{-}2}$, $T_{3\text{-}1}$ and $T_{3\text{-}2}$ performs extra computation for some solution points in the grid.

The implementation shown in Figure 4.9 should be applied to the tasks that are more expensive in terms of execution time. For unstructured grid applications, tasks T_1 and T_2 are more expensive than T_3 and T_4, therefore, it is reasonable to partition and overlap their

execution with data communication.

4.6 Experimental Results

CFD is a scientific area that analyze and solve problems involving fluid flows utilizing numerical approaches. Aerospace engineering is one of the fields that led the CFD development. The application utilized in this section solves the Navier-Stokes equations on unstructured grids utilizing a high order correction procedure via reconstruction methods. Details on this method are out of the scope of this document, for the further details refer to the paper by Wang [77].

The CFD application was originally implemented and optimized for running on CPUs. The most important optimization techniques utilized are loop unrolling, improved memory access and cache utilization.

In this section we present the results of the OpenCL implementation of the CFD application on a multi-GPU system utilizing the parallel techniques proposed in this chapter. The results presented correspond to a grid with 447,944 cells, 21 solution points per cell and 4 variables per solution point. The multi-GPU system utilized is composed by two GPUs NVidia Tesla C2070 with 448 cores and 6GB of global memory, and an Intel Xeon 2.67GHz hex-core processor.

First, the percentages of execution time for every stage in the main loop are given. These percentages are important because they hint on which task to parallelize. The percentages were computing by running small problems (around 20,000 cells) on the OpenCL program in one GPU.

Table 4.2: Time distribution of tasks of Algorithm 2

Task	$\mathbf{T_1}$	$\mathbf{T_2}$	$\mathbf{T_3}$	$\mathbf{T_4}$
Percentage	47.1%	42%	10.7%	0.2%

Table 4.2 presents the time distribution for each stage. As it was expected the neighbor analysis and local cell analysis stages, tasks T_1 and T_2 respectively, account for 89% of the execution time of one loop iteration, while the local variable update and boundary condition variables update stages, tasks T_3 and T_4 respectively, account for the remaining 11%. This time distribution is consistent with the type of computation performed by every stage. It is

important to say that while these percentages can slightly change for bigger problems, the trend will be the same.

As it was mentioned in the Section 4.5, the PCIe interconnection network adds some time overhead due to the shared switch when several devices are trying to communicate data between them at the same time. Next, Table 4.3 shows the times for one-way, GPU-0→GPU-1, and two-way, GPU-0↔GPU-1, communication between GPUs, as well the overhead introduced.

Table 4.3: Overhead introduced by the PCIe interconnection network in simultaneous data transfers

Size	**One-way (msec)**	**Two-way (msec)**	**Overhead (%)**
$\frac{1}{4} \times$ **Cells**	26.2	35.4	35.1%
$\frac{1}{2} \times$ **Cells**	52.9	71.5	35.2%
$\frac{3}{4} \times$ **Cells**	69.2	93.6	35.2%
Cells	86.3	116.7	35.2%

Table 4.3 shows the transfer time and switch overhead for communicating several fractions of the grid information. The size of the data is with respect to the number of cells in the grid, and it can be computed using the first equation on Table 4.1. It is interesting to notice, that the switch overhead is nearly constant at 35%, hence the expected time overhead due to simultaneous communications increases linearly with the grid size.

The workload of the grid is evenly divided in two GPUs (except for T_4, that it is not divided because of its amount of computation), and as expected the execution times of each half is very similar. The task execution times for half of the grid and for the summation of the two halves are shown in Table 4.4.

Table 4.4: Task execution times for the grid

Task	**T_1 (msec)**	**T_2 (msec)**	**T_3 (msec)**	**T_4 (msec)**
Half Grid	20.3	17.9	4.6	0.02
Full Grid	40.6	35.9	9.2	0.02

Next, from Tables 4.3 and 4.4 the results for the implementation of the different approaches

introduced in Sections 4.4 and 4.5 are presented in Table 4.5.

Table 4.5: Execution times for the proposed implementations

Implementation	**Computation**		**Communication**		**Total**	**Mem. Saving**
	(msec)	**(%)**	**(msec)**	**(%)**	**(msec)**	**(%)**
One GPU	85.72	47%	96.57	53%	182.29	48%
Task Parallelism	45.3	21%	172.6	79%	217.9	0%
Data Parallelism	43.0	38%	71.5	62%	114.5	41%
Data Par. C-C Ov.	43.0	39%	66.9	61%	109.9	41%
Cp-Cm Overlap	47.6	58%	34.2	42%	81.8	8%

Task parallelism, as expected, present the worst performance because of the high cost of data communication, near to 80% of the total time is dedicated to data transfer. Since the cost of communication is close to five times the cost of computation of T_2, i.e. 172.6 msec vs 35.9 msec, implementing computation-communication overlap will not have a big impact hiding the communication time.

Data Parallelism reduces the total execution time by about 50% compared to the task parallelism case. However, the communication time still accounts for more than 60% of the total time, which according to Figure 4.7 means that the GPUs remain idle during that time.

The last approach proposed, based on computation-communication overlap, reduces the total execution time by about 28% compared to the data parallelism case, and increases the time that GPUs are busy by about 20% compared to the data parallelism case. It can be noticed from Table 4.5 that by overlapping communication with computation, 48% of communication time is hidden by computation, which translates to more efficient utilization of computational resources.

As a reference and for comparison purposes, Table 4.6 presents the execution times of the original C/C++ implementation, the OpenCL implementation running in a CPU, one GPU and the computation-communication overlap approach running in two GPUs. The execution times shown correspond to the application running 1,000 iterations.

When the OpenCL implementation is executed on the hex-core CPU, all the cores are used,

Table 4.6: Execution times for 1,000 iterations on one CPU and two GPUs

Implementation	# Dev.	Time (sec)	Speedup
CPU (C/C++)	1	3,560	1.0
CPU (OpenCL)	1	1,298	2.7
GPU (OpenCL)	1	336	10.6
GPU (OpenCL)	2	270	13.2

whereas the original C/C++ implementation uses only one core because is single threaded, which is reflected on the speedup of 2.7 achieved. No higher speedup is achieved in the hex-core CPU because the application is not tuned for running in this architecture, the main purpose of presenting this speedup is to demonstrate that an OpenCL application tuned for GPUs can run on other architecture/platforms with no changes.

It is important to notice that the two-GPU implementation achieves approximately a 24% speedup with respect to the single-GPU implementation. This is due to the increased communication overhead, represents 42% of the total computation time. When efficient communication technologies, or OpenCL library improvements, emerge, the communication overhead will be reduced and the speedup will improve.

4.7 Conclusions

In this paper we proposed three different schemes for parallel execution in multi-GPU systems. In terms of added complexity to the application logic, task parallelism adds less complexity, however, the amount of task parallelism achievable is limited by the data dependencies among tasks.

Data parallelism achieves better performance than task parallelism, however, depending on the application logic, i.e. data dependencies, requires to transfer data among tasks executed on different GPUs.

Computation-communication overlapping combined with data parallelism achieves the best performance, however, adds more complexity than the two previous approaches. The proposed

scheme that utilizes computation-communication, not only hides communication overhead, but also decreases the switch overhead on the PCIe interconnection network by not having both GPUs transmiting data at the same time.

In general, it can be noticed that data communication among GPUs limits the achievable performance improvement. In current multi-GPU systems, the data exchange is performed using pinned memory through the host, which imposes the high overhead observed in the results. In the future, faster communications hardware, and proper software support, will help to reduce the communication overhead, and by this way our solution will achieve higher speedup.

CHAPTER 5. Kernel-Driven Data Analysis of GPU Applications

5.1 Introduction

Our framework for enhancing a single-GPU OpenCL application through program transformation relies on the analysis of the application source code to obtain the data dependencies among kernels and access patterns. This chapter starts by introducing the framework, and then the techniques utilized for source code analysis are presented.

5.2 An OpenCL Application

An OpenCL application is composed of pieces of code that are executed by the CPU of the system, i.e. host code, and also by pieces of code that are executed by an accelerator (GPU, FPGA, etc), i.e. kernel code or simply kernel.

In general, a scientific application is a sequence of kernels that are executed sequentially and might be repeated until a condition is satisfied or a number of iterations is reached. Algorithm 3 shows a group of N_K kernels executed sequentially, with $N_K - S$ kernels executed repeatedly inside a loop. In between two consecutive kernels, host code can be executed to initialize kernel parameters, copy information back from global memory to main memory or vice versa.

Enhancing a single-GPU OpenCL application for providing multi-GPU execution support or application-level checkpointing support at source code level implies program transformation. The host and kernel source code are modified with constructions that support the desired feature, e.g. constructions for data and computation decomposition, data transfers, etc.

Program transformation can be done manually, which requires knowledge of the application, or automatically through a pre-compiler tool that analyzes the application and enhances the source code with constructions that support the desired feature.

Algorithm 3 A scientific application

```
1: Initialize initial state
2: Kernel 0 (input variables, output variables)
3: ...
4: Some host code
5: Kernel S − 1 (input variables, output variables)
6: counter ← 0
7: repeat
8:    Kernel S (input variables, output variables)
9:    Some host code
10:   Kernel S + 1 (input variables, output variables)
11:   ...
12:   Some host code
13:   Kernel N_K − 1 (input variables, output variables)
14:   Update exit condition variables
15:   counter ← counter + 1
16: until ( exit = true ) or ( counter = Max )
17: Finishing code
```

Next, we propose a framework for enhancing single-GPU OpenCL applications to support multi-GPU execution or application level checkpointing.

5.3 Framework for Enhancing Single-GPU OpenCL Applications

Our framework for enhancing single-GPU applications has two stages: first, the kernel and host code are analyzed, and then this source code is transformed into one that supports either multi-GPU execution or application-level checkpointing. Figure 5.1 shows the two main stages for transforming single-GPU applications as well as the inputs required and outputs produced.

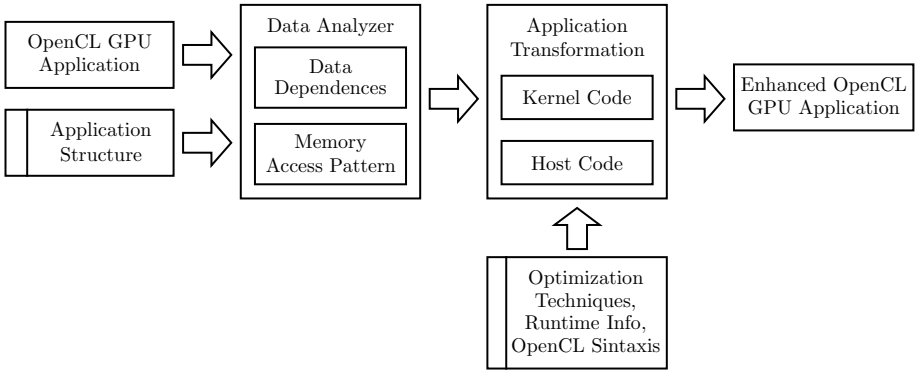


Figure 5.1: Main stages for transforming a single-GPU application

This chapter describes the *Data Analyzer* stage of the framework, its inputs required and the outputs produced, the *Application Transformation* stage is described in detail in Chapters 6 and 8.

The major goal of the data analyzer is to provide information that supports the program transformation. The information required to transform an OpenCL application into one that supports multi-GPU execution or application level checkpointing is data dependencies among kernels and data access patterns for each array stored in the memory of the GPU.

In the case of providing multi-GPU execution, the information provided by the data analyzer allows to determine (i) data distribution, (ii) data transfers required, and (iii) computation decomposition; whereas in the case of providing application-level checkpointing, that information allows to determine (i) the application state, (ii) checkpoint location, and (iii) whether it is possible to overlap checkpoint operations and kernel execution.

5.4 Application Structure Information

As shown in Figure 5.1, along with the host and kernel source code, the data analyzer requires information about the structure of the application. Application structure information simplifies the design and implementation of the data analyzer as it provides the file location of the kernel body declaration, subroutine names where the kernel parameters are initialized and the kernels are executed. This information is classified in two types: (i) general declarations, such as file location, and (ii) kernel information that includes information for every kernel defined in the OpenCL application, such as name, execution sequence, etc.

In this work, the application structure information is represented using XML language. The XML language allows to represent the application structure easily and intuitively through a small set of tags.

Next, Table 5.1 shows the tags, and their description, utilized for describing the structure of an OpenCL application.

To illustrate the usage of the XML tags presented in Table 5.1, Listing 5.2 shows the XML representation of the structure of the OpenCL application of Listing 5.1, which is composed of two kernel functions and the host code.

Table 5.1: Tags for describing an OpenCL application

Tag	Description
kernel	Kernel specification
loop	Indicates if a kernel is inside of a loop
parallelize	Indicates if kernel parallelization is allowed
path	Location of the files
file	File name
name	Name of a kernel or subroutine
init	Subroutine where the kernel is initialized
caller	Subroutine where the kernel is executed

File: kernel.cl

```
1  __kernel void VectorAdd(__global int* c, __global int* a, __global int* b, int coefficient)
2  {
3    // Index of the elements to add
4    int n = get_local_id(0) + get_local_size(0) * get_group_id(0);
5    unsigned int m = 2 * n;

7    // Sum the nth element of vectors a and b and store in c
8    c[n] = ( a[m] + b[m] ) * coefficient;
9  }

11 __kernel void VectorSub(__global int* c, __global int* a, __global int* b, __global int* geometry)
12 {
13   // Index of the elements to compute
14   unsigned int n = get_global_id(0);
15   int m = geometry[n];

17   // Substract neighbor elements and scale it properly
18   c[n] = ( a[m] - b[m] ) * c[n];
19 }
```

File: main.c

```
20 int main(void)
21 {
22   // Initialize the OpenCL environment: context, queues
23   ...
24   // Create kernel objects
25   VectorAddObj = clCreateKernel(..., ''VectorAdd'', ...);
26   VectorSubObj = clCreateKernel(..., ''VectorSub'' ...);
27   // Set parameters for kernels VectorAdd and VectorSub
28   clSetKernelArg(VectorAddObj, 0, ..., (void*)&arrayC);
29   clSetKernelArg(VectorAddObj, 1, ..., (void*)&arrayA);
30   clSetKernelArg(VectorAddObj, 2, ..., (void*)&arrayB);
31   clSetKernelArg(VectorAddObj, 3, ..., (void*)&coefficient);
32   clSetKernelArg(VectorSubObj, 0, ..., (void*)&arrayC);
33   clSetKernelArg(VectorSubObj, 1, ..., (void*)&arrayA);
34   clSetKernelArg(VectorSubObj, 2, ..., (void*)&arrayB);
35   clSetKernelArg(VectorSubObj, 3, ..., (void*)&geometry);

37   // Send VectorAdd for execution
38   clEnqueueNDRangeKernel(queue, VectorAdd, ...);
```

```
39    // Send VectorSub for repeated execution
40    do {
41      clEnqueueNDRangeKernel(queue, VectorSub, ...);
42    } while ( !ExitCondition );
43    return 0;
44  }
```

Listing 5.1: Kernel and host code for an OpenCL application with two kernels

Listing 5.1 includes the body specification for two kernels, i.e. `VectorAdd` (lines 1 to 9) and `VectorSub` (lines 11 to 19), in the file `kernel.cl`, and the body specification of the `main` subroutine (lines 20 to 44) in the file `main.c`. It is important to notice that the parameter initialization and kernel execution are in the subroutine `main()` in the main.c file, lines 28 to 35 and lines 38 and 41, respectively.

```
1  <configuration>
2     <path>./kernels</path>
3     <kernel>
4        <file>kernel.cl</file>
5        <name>VectorAdd</name>
6        <caller>
7           <file>main.c</file>
8           <name>main</name>
9        </caller>
10       <init>
11          <file>main.c</file>
12          <function>main</function>
13       </init>
14    </kernel>
15    <kernel>
16       <file>kernel.cl</file>
17       <name>VectorSub</name>
18       <loop>true</loop>
19       <caller>
20          <file>main.c</file>
21          <name>main</name>
22       </caller>
23       <init>
24          <file>main.c</file>
25          <function>main</function>
26       </init>
27    </kernel>
28 </configuration>
```

Listing 5.2: An example of a XML application structure file

The general declarations section, line 2 in Listing 5.2, defines the location of the source files. The kernel specification section, lines 3 to 27 in Listing 5.2, includes information for the two kernels defined in the OpenCL application of Listing 5.1. Table 5.2 summarizes the information presented in the XML structure file listed in Listing 5.2.

Table 5.2: Summary of the OpenCL application structure represented in Listing 5.2

Description		VectorAdd		VectorSub	
		Value	**Line**	**Value**	**Line**
File		kernel.cl	4	kernel.cl	16
Execution Sequence		1	-	2	-
Inside of a loop		No	-	Yes	18
Caller	Subroutine	main	8	main	21
	File	main.c	7	main.c	20
Init	Subroutine	main	12	main	25
	File	main.c	11	main.c	24

Although the execution order is not explicitly specified in the XML structure file, it is implicitly defined by the order in which the kernels are listed in the XML structure file.

5.5 The Set of Kernel Structure Lists

The application structure information provided by the XML structure file is used by the kernel analyzer to access the appropriate files and to process the subroutines and kernels specified. This information is used throughout all the steps in the data analysis and program transformation.

To generate the data usage, access patterns and data dependencies, the data analyzer creates and maintains a set of kernel structure lists, which contains information of every array utilized by each kernel.

This set of lists has one list per kernel, i.e. a kernel structure list, where every list is generated by inspecting the kernel declaration, and it contains the names of the arrays stored in GPU memory, i.e. global or constant memory, that are utilized by the kernel. Each kernel structure list is described as: kernel<$array_0$:$properties_0$, ..., $array_n$:$properties_n$>, where *kernel* and $array_i$ are self-describing, and $properties_i$ contains the usage, access pattern, host variable name and dependencies information for $array_i$: $properties_i$={$usage_i$, $access_i$, $name_i$, $dependency_i$}. For instance the kernel structure list for VectorAdd in Listing 5.1 is: VectorAdd<c, a, b>. It is important to emphasize that only arrays are included in the lists,

for example the list correspondent to VectorAdd does not include the scalar parameter, i.e. coefficient.

5.6 Data Usage and Access Patterns

This information is obtained by analyzing the body of each kernel. From now on the terms array and data are used interchangeably to refer to an array stored in GPU memory.

Data usage is key to support the data dependencies analysis as it indicates if an array is utilized by a kernel as an input, output or both. At each occurrence of the array name in the kernel body, the usage in the kernel structure list is updated appropriately.

The data access patterns can be classified as linear and non-linear (random). The major goal of determining the data access patterns for each array is to determine if the data access range of a work-group is bounded to a subset of the data and the subsets are non-overlapping; otherwise the data access pattern is referred to as unbounded. Equation 5.1a presents the constraint required for bounded access and Equation 5.1b for non-overlapping data sets, where N is the number of partitions.

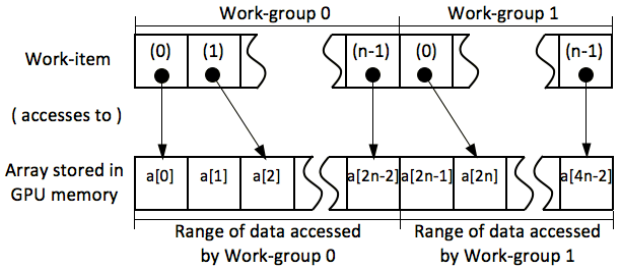
$$\bigcup_{i=1}^{N} array_i = array, \quad \forall\, array_i \subset array \tag{5.1a}$$

$$\bigcap_{i=1}^{N} array_i = \emptyset \tag{5.1b}$$

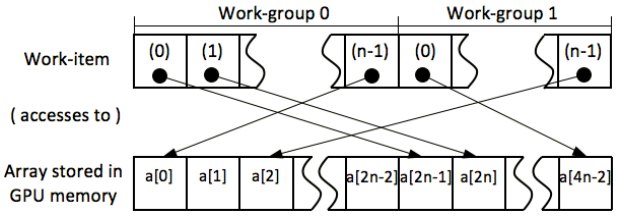
Access patterns are crucial for defining data and computation decomposition that allows to overlap kernel execution and checkpoint operations, and, along with the data dependences, for identifying the application state at each kernel. In particular, to overlap kernel execution and checkpoint operations, all its output arrays need to satisfy both constraints defined in Equation 5.1.

For instance, Figure 5.2a shows the case where the data accessed by two work-groups is limited to two non-overlapping subsets, while Figure 5.2b shows the case where the ranges of data accessed cannot be limited to subsets.

In the OpenCL programming model, a kernel defines the data access and computations for a single work-item, and then those features can be generalized to a work-group. Therefore, the



(a) Linear (Bounded) data access



(b) Random (Unbounded) data access

Figure 5.2: Array (memory) access pattern

data analyzer analyzes the access pattern of a single work-item and then it generalizes to the work-group level. Determining the access pattern is achieved by analyzing the index utilized for accessing data stored in arrays in GPU memory, which simplifies the analysis to determining if an index is a linear combination of the work-item and/or work-group identifiers.

OpenCL identifies a work-item using a non-unique local identifier, i.e. local id, and an unique global identifier, i.e. global id [82]. The work-item local identifier is unique inside a work-group, but it is non-unique among two different work-groups, for instance, in Figure 5.2 the work-items are identified by their local identifier, which is repeated in the two work-groups presented. The work-item global identifier is unique, regardless the work-group that the work-item belongs to. This global id can be represented as a linear combination of the local id and the group id, for instance, the indexes computed in lines 4 and 14 of Listing 5.1 are equivalent.

A linear combination of the global id, or an equivalent expression, guarantees bounded non-overlapping access, hence, the kernel analyzer verifies that the arrays are indexed using a linear combination of the global id, see Equation 5.2.

$$index = \alpha \times global\_id(0) + \beta, \quad \forall\, \alpha, \beta \in \mathbb{Z} \ \text{ and } \ \alpha \neq 0 \tag{5.2}$$

Although OpenCL supports multidimensional kernels, i.e. multidimensional arrays of work-items, in this work only one dimensional kernels are supported, hence, global id refers to a unidimensional identifier.

Indexes based on dynamic values, such as values in other arrays, are considered non linear, and hence the access pattern is unbounded. This scenario is shown in line 15 of Listing 5.1, where the index m is computed using values from an array.

5.7 Data Dependencies

The data dependencies analysis is performed in two steps: first, it parses the host code to collect the variable names of the kernel objects and kernel parameters, and then it traces the variable names of the parameters across kernels.

Although the kernel structure list include the kernel name and the name of its parameters, it is necessary to parse the host code for collecting the names of the kernel objects and variables passed as parameters. The XML structure file provides information of the subroutine, i.e. name and location, where the kernel object and parameters are initialized, see lines 10 to 13 and 23 to 26 in Listing 5.2.

Kernel objects, utilized for passing parameters to kernel functions as well as for executing kernels, are defined in the host code using the OpenCL function `clCreateKernel()`.

In OpenCL kernel parameters are passed using the function `clSetKernelArg()`, which requires a kernel object, the sequence number and variable name of the parameter, among other parameters. The parameter sequence number refers to the order in which the parameters were declared on the kernel.

The data analyzer utilizes the kernel object and the parameter sequence number to obtain the variable name of the parameters and to update the proper kernel structure list.

After the data usage, access pattern and dependence analysis is performed, the updated kernel structure list for VectorAdd is: VectorAdd<c:{output, bounded, arrayC}, a:{input, bounded, arrayA}, b:{input, bounded, arrayB}>, and for VectorSub is: VectorSub<c:{input/output, bounded, arrayC}, a:{input, unbounded, arrayA}, b:{input, unbounded, arrayB}, geometry:{input, bounded, geometry}>.

To determine the transformation details it is important to understand the data flow among the kernels, and, hence, the data dependencies among them. The data dependencies are presented using a Dependence Graph, where the kernels are represented as nodes and the dependencies as directed arrows. Data dependencies among kernels are represented by $K_i \rightarrow K_j$, which denotes that K_j depends on data produced by K_i.

Besides information about kernels parameters, data usage and execution order, loop information is important, because kernels that are inside a loop can potentially have dependences from previous iterations, which is not the case for kernels that are outside a loop. Hence, the first step performed by the data analyzer towards defining data dependencies is to find which kernels are outside and inside a loop.

As shown in Algorithm 3, a scientific application is a set of N_K kernels sequentially executed, with S kernels executed before the main loop and $N_K - S$ kernels executed repeatedly inside a loop. Hence, finding the kernels that are outside the loop is reduced to finding the value of S.

Once the kernel analyzer has obtained the loop information, it proceeds to find data dependencies among kernels. Finding data dependencies requires searching relationships between the inputs and outputs of different kernels.

Algorithm 4 presents details for the main steps taken for performing data dependencies analysis. This algorithm requires two inputs: the set of kernel structure lists and the number of kernels, which are represented by the array *kernel* and the variable N_K, respectively.

The first section in Algorithm 4 finds the start of the loop, if any, by taking each kernel and checking if it is inside a loop, i.e. checking the *loop* flag in each kernel structure list. The next section, finds data dependencies by taking the input parameters of each kernel (k_i), starting from the the last until the first kernel, and looking for a match with the output parameters of the previous kernels (k_k); this procedure continues for each input parameter of k_i until a match is found or the last possible kernel is reached. The last possible kernel is represented in Algorithm 4 by the variable *lowerLimit*, which it can be either the first kernel, i.e. k_0, when the kernel k_i is not in a loop or k_{i+1} when the kernel k_i is in a loop. When a match is found, the dependency $k_k \rightarrow k_i$ is created.

Figure 5.3 presents the data dependencies between five kernels executed sequentially inside

Algorithm 4 Finding data dependencies

```
1: {This section finds the start of the loop}
2: i ← 0
3: loopStart ← −1
4: repeat
5:    if kernel[i].insideLoop() then
6:       loopStart ← i − 1
7:    end if
8:    i ← i + 1
9: until loopStart ≥ 0 or i = N_K
10: loopStart ← loopStart + 1
11: {Now finds the data dependencies}
12: for i = (N_K − 1) → 0 do
13:    for all input parameters in kernel[i] do
14:       current_input ← current parameter
15:       k ← i − 1
16:       lowerLimit ← 0
17:       if i = loopStart and kernel[i].insideLoop() then
18:          k ← N_K − 1
19:          lowerLimit ← i + 1
20:       end if
21:       found ← FALSE
22:       while not found and k ≥ lowerLimit do
23:          for all output parameters in kernel[k] do
24:             if current parameter = current_input then
25:                Create: kernel[k] → kernel[i]
26:                found ← TRUE
27:             end if
28:             if k = loopStart and kernel[i].insideLoop() and i < N_K − 1 then
29:                k ← N_K
30:                lowerLimit ← i + 1
31:             end if
32:             k ← k − 1
33:             {If no match was found inside the loop}
34:             if 0 ≤ k < lowerLimit and not found then
35:                k ← loopStart − 1
36:                lowerLimit ← 0
37:             end if
38:          end for
39:       end while
40:    end for
41: end for
```

a loop, where the solid arrows represent data dependences existing in the same loop iteration, and dotted arrows represent data dependences existing in the next loop iteration.

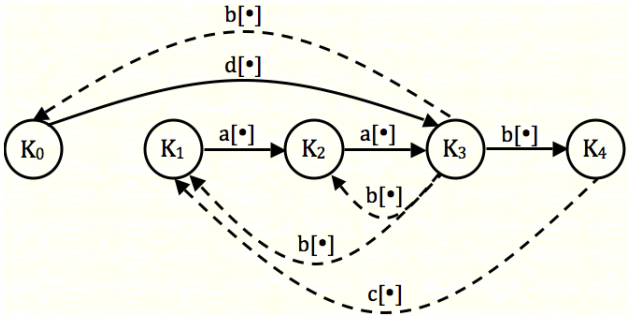


Figure 5.3: A Data Dependence Graph for an OpenCL application with five kernels

Next, Listing 5.3 presents the set of kernel structure lists generated by the kernel analyzer for the dara dependence graph of Figure 5.3. Although in general the kernel parameter name and variable name of the arrays might be different, in this example for simplicity they are the same.

1 K_0<loop, b:{input, bounded, b, K_3}, d:{output, bounded, d, *unused*}>
2 K_1<loop, b:{input, bounded, b, K_3}, c:{input, bounded, c, K_4}, a:{output, bounded, a, *unused*}>
3 K_2<loop, b:{input, bounded, b, K_3}, a:{input/output, bounded, a, K_1}>
4 K_3<loop, a:{input, bounded, a, K_2}, d:{input, bounded, d, K_0}, b:{output, bounded, b, *unused*}>
5 K_4<loop, b:{input, bounded, b, K_3}, c:{output, bounded, c, *unused*}>

Listing 5.3: The set of kernel structure lists for the Data Dependence Graph of Figure 5.3

The information provided in the set of lists presented in Listing 5.3 is utilized in the next stage of our framework to transform the application to support the desired feature.

CHAPTER 6. Parallelization of GPU OpenCL Applications

6.1 Introduction

Running single-GPU applications on multi-GPU systems can be approached by (i) providing libraries that parallelize a single GPU application at runtime [25, 26] and, (ii) transforming the application before compiling [27, 36].

Transforming an application implies source to source conversion, i.e. converting a single-GPU application into a multiple GPU application. Kim et. all present a framework for converting an OpenCL application into a multi-GPU CUDA application [27]; this framework shows to be effective for running applications, however, no dependence analysis is performed for determining data transfers requirements and also translating an application into CUDA restricts the usability of the framework to NVidia GPUs. In [36] a single-GPU CFD application written in OpenCL is manually instrumented for exploring different levels of parallelism and efficient techniques for reducing data transfer overhead.

While it is clear that transforming an application for running in multi-GPU systems is more efficient in terms of performance, manually transforming an application is time consuming and error prone.

In this chapter we propose a framework and implement a tool for automatic transformation of OpenCL applications written for running on single-GPU systems into an OpenCL application that runs on multi-GPU systems.

6.2 Parallelization

This section presents how the information provided by the kernel analyzer is utilized for program transformation. The data usage in combination with access patterns and the dependence

graph allows to determine parallelization details such as parallelism type, kernel (computation) decomposition, data distribution, data and kernel offsets, data transfer among GPUs and task synchronization.

The dependence graph helps to determine the type of parallelism that can be exploited, i.e. task or data parallelism. Although task parallelism is easier to implement, it does not guarantee load balancing nor reduces the application memory footprint required on a single GPU. Therefore, this work focuses on exploiting data parallelism.

6.2.1 Data Parallelism

Data parallelism requires to decompose the kernels in N partitions that can be executed concurrently. Kernel decomposition is achieved by evenly distributing work-groups among different GPUs. The number of partitions (N) cannot exceed the number of GPUs in the system (M). Equation 6.1 describes a kernel evenly decomposed into N kernel partitions, i.e. $kernel_i$, where the last kernel partition might be smaller than the others. The size of the kernel, i.e. $size(kernel)$, is expressed in terms of the number of work-groups.

$$\begin{aligned} size(kernel_i) &= \left\lceil \frac{size(kernel)}{N} \right\rceil \\ size(kernel_N) &\leq size(kernel_i), \quad 1 \leq i < N \leq M \end{aligned} \tag{6.1}$$

Distributing kernels among available GPUs is possible since the OpenCL execution model does not support synchronization among work-groups, hence, their execution is independent of each other. Figure 6.1 presents a case where n work-groups are evenly distributed among two GPUs.

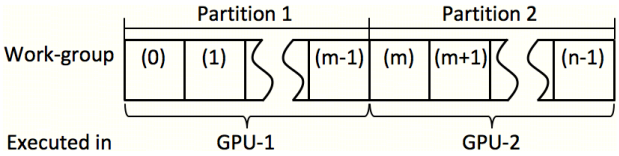


Figure 6.1: Kernel decomposition into two GPUs

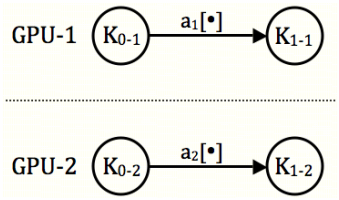
Work-groups are evenly distributed among the available GPUs, and if data access is linear then the data is evenly distributed as well. In this context, the number of elements and the

size of each data partition is given by Equations 6.2a and 6.2b.

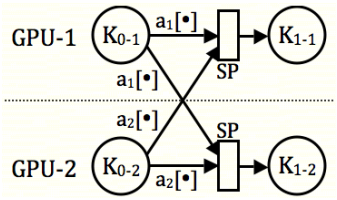
$$elements(array_i) = \frac{size(kernel_i)}{size(kernel)} \cdot elements(array) \tag{6.2a}$$

$$size(array_i) = elements(array_i) \cdot sizeof(data\_type) \tag{6.2b}$$

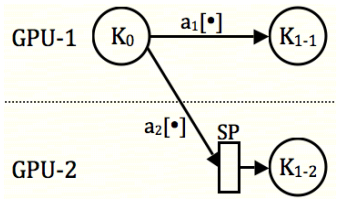
Data access patterns and dependencies among kernels limit the parallelism achieved and might impose data communication overhead due to data exchange among GPUs. For simplicity, Figure 6.2 presents for a system with two GPUs the possible data parallelism implementations of two kernels that have a data dependency, i.e. $K_0 \rightarrow K_1$, and different access patterns for the output and input of K_0 and K_1, respectively.



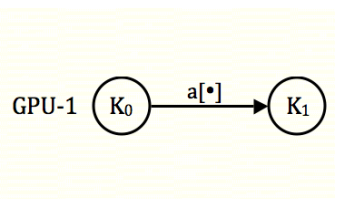
(a) O: Bounded, I: Bounded



(b) O: Bounded, I: Unbounded



(c) O: Unbounded, I: Bounded



(d) O: Unbounded, I: Unbounded

Figure 6.2: Data parallelism implementations for two kernels with a data dependency and different access patterns for the output and input

In Figure 6.2, partitions of kernel K_i and data $a[\cdot]$ are represented by $K_{i\text{-}j}$ and $a_j[\cdot]$, respectively, where j refers to the *j-th* data/kernel partition stored/executed in the *j-th* GPU.

The best and worst cases in terms of kernel decomposition and communication overhead for exploiting data parallelism are depicted in Figures 6.2a and 6.2d. The best scenario is when the data subsets accessed by every work-group meet the constraints defined in Equation 5.1, hence kernel decomposition with no communication overhead is possible. The worst scenario is

when neither of the kernels have linear access on the data, hence, although no communication overhead exist, no kernel decomposition is possible either. Figures 6.2b and 6.2c show scenarios where kernel decomposition imposes communication overhead, which will be discussed in Section 6.2.2.

It is important to highlight from Figure 6.2 that (i) non-linear data access impose data transfers between GPUs, (ii) synchronization points (SPs) are needed when data is transfered between GPUs, and (iii) no decomposition is possible when the kernel data access of the output array is non-linear.

6.2.2 Kernel Decomposition Performance

The performance of a multi-GPU application is determined by (i) the amount of computation available to keep the hardware busy and (ii) the communication overhead.

In this work it is assumed that there are enough work-groups in each kernel partition to keep the GPUs busy, hence, no performance loss due to kernel decomposition is assumed.

6.2.2.1 All to All Communication

Figure 6.2b presents the case where kernel decomposition imposes data exchange among GPUs due to the non-linear access of K_1 on the array $a[\cdot]$.

For a general case with N partitions, the amount of information exchanged, as a function of the number of elements is given by Equation 6.3.

$$\begin{aligned} data\_exchanged &= \sum_{i=1}^{N} (N-1) \cdot elements(array_i) \\ &= (N-1) \cdot elements(array) \end{aligned} \tag{6.3}$$

From Equation 6.3 the execution time for the parallelized kernels can be approximated as indicated in Equation 6.4.

$$\begin{aligned} time(\text{multi-GPU}) &= \frac{time(\text{single-GPU})}{N} + time(data\_exchanged) \\ &= \frac{time(K_0) + time(K_1)}{N} + (N-1) \cdot time(array) \end{aligned} \tag{6.4}$$

As it can be seen from Equation 6.4, kernel decomposition, i.e. the number of kernel partitions N, has a dual effect, both favorable and adverse, on the execution time of a multi-GPU applica-

tion: (i) the computation time is linearly reduced by a factor of N, and (ii) the communication overhead increases with the number of partitions, i.e. the number of GPUs.

6.2.2.2 One to All communication

Figure 6.2c shows a scenario where a non-linear access pattern not only imposes communication overhead, but also no kernel decomposition is possible.

In this case the amount of data exchanged and execution time is given by Equations 6.5 and 6.6.

$$data\_exchanged = (N-1) \cdot elements(array_i) \tag{6.5}$$

$$time(\text{multi-GPU}) = time(K_0) + \frac{time(K_1)}{N} + (N-1) \cdot time(array_i) \tag{6.6}$$

Similarly to the previous case, the computation scales linearly, whereas the communication overhead does not scale as the number of GPUs increases.

6.3 Program Transformation

Transforming a single-GPU OpenCL application into a multi-GPU OpenCL application is described in this section as a five step process:

- Decomposing the kernel at kernel code level.
- Adding multiple device support to contexts.
- Adding support for multiple device execution.
- Transferring data to satisfy dependencies.
- Adding support for communication-computation overlapping

6.3.1 Decomposing the Kernel at Kernel Code Level

As defined in the previous section, kernel decomposition is achieved by distributing the work-groups among the available GPUs. In this context, the number and size, i.e. number of

work-groups, of kernel partitions are given by the number of devices existing in the system, unless otherwise defined in the host code.

Data and kernel decomposition modify the data indexes, as well as the global and work-group identifiers, therefore, depending on the data access pattern of a kernel, its code might need to be transformed into one that supports decomposition. For a kernel with non-linear access, an offset is required to properly access the data. As shown in Equation 8.9, this offset is computed based on the kernel partition number and in the number of work-groups per kernel partition, where the size of a kernel partition was defined in Equation 6.1.

$$offset_i = (i-1) \cdot \left\lceil \frac{size(kernel)}{N} \right\rceil \tag{6.7}$$

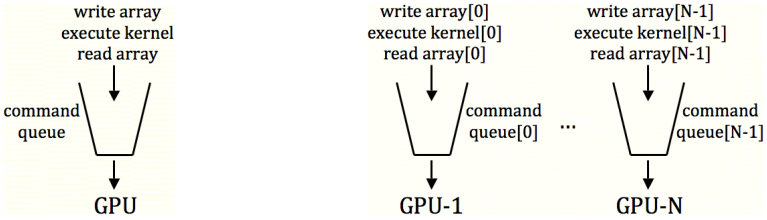
The kernel is transformed by adding a new parameter to the declaration that receives the offset, and modifying the index of the arrays that have non-linear access using the offset: $n = global\_id(0) + offset \cdot get\_local\_size(0)$.

6.3.2 Adding Multiple Device Support to Contexts

As introduced previously, an OpenCL context includes devices and objects for kernels, memory and command queues. The host program places commands in a command queue, which are scheduled for execution on the device that is associated with the queue. Hence, a context requires at least one command queue associated with a device.

Providing support for multiple GPUs requires to define and associate separate queues to the GPUs that will receive commands. In addition to separate queues, GPUs require their own memory and kernel objects, hence, these objects need to be redefined as object arrays and associated to GPUs.

Figure 6.3 shows the execution of a single-GPU and multi-GPU applications, emphasizing that each GPU requires its own object, i.e. kernel, memory and queue, represented as an array.



(a) Single-GPU app.

(b) Multi-GPU app.

Figure 6.3: Context with support for single-GPU and multi-GPU

6.3.3 Adding Support for Multiple Device Execution

The host orchestrates the execution of kernels, which includes placing commands to transfer data to/from a GPU, set kernel parameters and execute a kernel. For multi-GPU applications the host code performs these operations for each GPU, with the difference that kernels and data are decomposed. To support data and kernel decomposition and depending on the data and kernel decomposition details, the host code: (i) computes offsets and sizes for data access, (ii) set appropriate kernel parameters, and (iii) computes kernel sizes.

The host transfers the appropriate pieces of data to/from each GPU based on an offset and a data size for each array, which are computed as shown in Equations 6.7 and 6.2b.

Additionally, the host places commands in the command-queues to assign the appropriate memory objects and offsets to the kernel parameters.

Finally, the kernel partition size is computed, as in Equation 6.1, and passed with the command to execute the kernel.

6.3.4 Transferring Data between GPUs to Satisfy Dependencies

As introduced in Section 6.2, data dependencies and access patterns might introduce data transfers between GPUs. The host code is enhanced with commands to transfer data and to synchronize execution among different command-queues.

Data transfer between GPUs is performed in two steps: first, from GPU memory to the CPU memory, and then to the other GPU. Since, the two commands for transferring data

among GPUs are placed in different command-queues, synchronization between them is needed. Synchronization among command-queues is achieved through OpenCL events.

6.3.5 Adding Support for Communication-Computation Overlapping

To support communication-computation overlapping further kernel and data decomposition is required at every GPU. In this work we decompose into two kernels and data subsets.

In terms of structures required to support this overlapping, a new command queue associated with every GPU is required. This additional command queue is utilized for concurrent kernel execution and memory operations.

6.4 Experimental Results

In this section we present the results for the parallelization of two applications utilizing the tool described in this chapter. Interesting applications for demonstrating the tool contain kernels with data usage and data dependencies as depicted in Figures 6.2a and 6.2b.

The first application performs lineal algebra operations: $E = A + B + C \times D$, which for demonstration purposes has been divided into two kernels: K_0 that computes $E = A + B$ and K_1 computes $E = E + C \times D$. Kernels are organized in such a way that each element of the resulting vector is computed by a kernel work-item. It is clear that a dependency exists between K_0 and K_1 and due to the organization of the computations the memory access pattern of both kernels is linear, therefore, this application corresponds to the case shown in Figure 6.2a, where data parallelism can be exploited with no data exchange among GPUs.

The second application chosen is a CFD application that solves the Navier-Stokes equations on unstructured grids utilizing a high order correction procedure via reconstruction methods. We omit details of the method used for the application, for further details refer to the paper by Wang [77]. Implementation of this method on single-GPU and multi-GPU systems has been studied in Chapters 3 and 4. This application is composed of four kernels that perform neighbor analysis (K_0), cell analysis (K_1), local variables update (K_2) and boundary conditions update (K_3). From the analysis in previous chapters, the data dependencies and usage of the two last kernels correspond to the case shown in Figure 6.2b.

Single-GPU versions of these two applications are provided as input to the tool for obtaining multi-GPU versions.

6.4.1 Multi-GPU & Network Performance Issues

The multi-GPU system utilized for testing and benchmarking the applications is composed by four GPUs NVidia Tesla C2070 with 448 cores and 6GB of global memory, and an Intel Xeon 2.5GHz quad-core processor, see Figure 6.4.

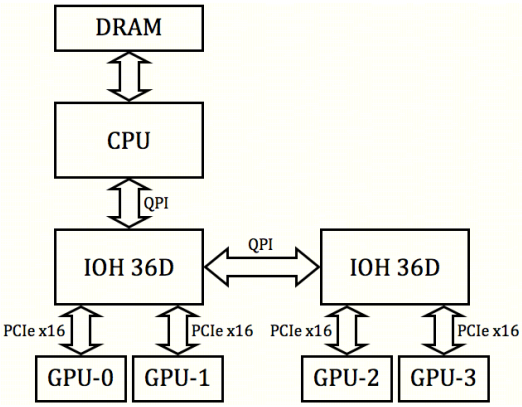


Figure 6.4: Architecture of the 4-GPU system used

Our system implements two I/O Hubs (IOHs), which introduces non-uniform memory access (NUMA); notice that GPU-2 and GPU-3 must traverse an additional QPI link. In addition to NUMA effects, concurrent data transfers increase the contention on shared system resources. Both, NUMA and contention degrade the effective bandwidth achieved by the GPUs. This bandwidth degradation has serious implications on achievable performance, as it will be shown later in this section.

Next, Table 6.1 presents the bandwidth measured for concurrent data transfers (2, 3 or 4 GPUs), and, for comparison purposes, for non concurrent transfers (1 GPU).

Table 6.1: PCI Express bandwidth degradation due to concurrent communication and NUMA effects

Concurrent Transfer	**GPU-0 (GB/s)**	**GPU-0,1 (GB/s)**	**GPU-0,1,2 (GB/s)**	**GPU-0,1,2,3 (GB/s)**
Host to Device	5.6	5.2	3.5	2.6
Device to Host	6.1	3.9	2.5	1.8

Table 6.1 shows an almost linear PCI bandwidth degradation with respect to the number of GPUs concurrently transferring data. Hence, it can be concluded that it is expected more communication overhead when more GPUs exchange data.

6.4.2 Performance Results

The results for the linear algebra application are summarized in Table 6.2. As expected, almost linear speedup is achieved because no intermediate results are exchanged among GPUs.

Table 6.2: Execution time for the linear algebra application

GPUs	K_0 (msec)	K_1 (msec)	Time (msec)	Speedup
One	18.86	27.02	46.03	1.00
Two	9.45	13.53	23.21	1.98

The results for the CFD application are presented in Table 6.3, which correspond to a grid with 447,944 cells.

Table 6.3: Execution times for single and multiple GPUs implementations of the CFD application

GPUs	Implementation	K_0 (msec)	K_1 (msec)	K_2 (msec)	K_3 (msec)	Comm. (msec)	Total (msec)	Speedup
One	Large Data Set	49.45	34.28	9.2	0.02	77.61	170.60	1.00
Two	No Overlapping	24.75	17.16	4.6	0.02	50.79	97.32	1.75
Two	K_2 & K_3 Overlap	24.75	17.16	4.6	0.02	46.30	92.83	1.84
Two	K_0 & K_1 Overlap	24.75	17.16	9.2	0.02	31.62	82.75	2.06
Four	No Overlapping	12.39	8.6	2.32	0.02	70.88	94.21	1.81
Four	K_2 & K_3 Overlap	12.39	8.6	2.32	0.02	69.89	93.22	1.83
Four	K_0 & K_1 Overlap	12.39	8.6	9.2	0.02	58.69	88.90	1.92

This table includes one, two and four GPUs implementations with and without computation-communication overlapping. The multi-GPU non-overlapping implementations utilized to produce the results presented were not produced by the tool described in this work, they are included for completeness to have a reference of the communication overhead imposed.

The single GPU implementation assumes that the data does not fit in memory, hence, it needs to load/unload data into the GPU memory as needed. This implementation is utilized as a baseline for comparing to the implementations with two and four GPUs. Besides the non-overlapping implementation, other two implementations are presented: (i) one where K_2 and K_3 are utilized for computation-communication overlapping, and (ii) another where K_0 and K_1 are utilized for computation-communication overlapping (this implementation required some manual changes). These two implementations present different speedups due to the overlapping achieved. The implementation where K_0 and K_1 are utilized for overlapping present higher speedup for two and four GPUs because of their increased computation time.

As it can be seen in Table 6.3, the communication time is bigger than the computation time in both cases of parallelization with two and four GPUs, hence, it is crucial to hide as much communication overhead as possible by overlapping it with kernel computation. However, in this application not all the communication overhead can be hidden by computation, up to approximately 40% of the communication overhead still can be hidden by computation on the GPUs.

Although the two-GPU implementation presents almost linear speedup, the communication overhead is still high, it accounts for about 38% of the total time in the best case.

The case with four GPUs require more attention, because the performance obtained is comparable with the performance of a two-GPU implementation, hence, no linear speedup is achieved. The main reason for this low performance is the increased communication overhead, it accounts for approximately 75% of the total time presented for the non-overlapped case.

In a multi-GPU implementation of the CFD application, decomposition scales linearly the computation time with a factor of N, however, the communication overhead does not scale, instead it increases due to (i) the larger amount of information that needs to be exchanged among GPUs, see Equation 6.3, (ii) the inability to perform direct transfers among GPUs, and (iii) the PCI Express bandwidth degradation due to concurrent memory operations. As presented on Table 6.1, the PCI Express bandwidth degrades almost linearly with a factor of N for concurrent communication. Besides the bandwidth degradation for concurrent communications, OpenCL support for data exchange among devices is limited, information exchange has to be performed

using a host buffer, which involves a read and a write operation, `clEnqueueReadBuffer()` and `clEnqueueWriteBuffer()`. When better hardware communication mechanisms among multiple GPUs will emerge, our methods will deliver much more effective solutions.

6.4.3 Nvidia GPUDirect

GPUDirect is a feature introduced by Nvidia for its Fermi GPUs. GPUDirect allows direct data exchange among GPUs connected to the same IOH with no host buffer involved. This feature allows for faster data exchange and reduced bandwidth degradation for concurrent communication. Next, Table 6.4 presents the bandwidth achieved for data exchange using a host buffer and using GPUDirect.

Table 6.4: Bandwidth for communications using a host buffer and GPUDirect

Implementation	**GPU0 → GPU1 (GB/s)**	**GPU0 ↔ GPU1 (GB/s)**
Using Host Buffer	2.95	2.18
GPUDirect	4.95	4.65

As it can be seen in Table 6.4, GPUDirect achieves a higher bandwidth and lower bandwidth degradation for concurrent transfers.

Next, Table 6.5 presents a projection for the communication overhead and speedup achieved in the CFD application using GPUDirect.

Table 6.5: Projection of communication times for the CFD application using GPUDirect

GPUs	**Implementation**	**Comp. (msec)**	**Comm. (msec)**	**Total (msec)**	**Speedup**
One	Large Data Set	92.95	77.61	170.60	1.00
Two	No Overlapping	46.53	≈ 25	71.53	2.39
Two	K_2 & K_3 Overlap	46.53	≈ 20	66.53	2.56

GPUDirect clearly helps to reduce the communication overhead, and by this way higher speedups can be achieved. However, at the time of this writing no GPUDirect support for OpenCL is provided by Nvidia.

6.5 Conclusions

In this chapter we described a framework and implemented a tool for parallelizing single-GPU OpenCL applications. Besides the single-GPU application, the proposed method requires as input a XML file that describes information about the application structure.

Data dependence, data usage and access pattern analysis are key for determining data and computation decomposition, as well as for determining data transfers among GPUs required for satisfying data dependencies.

Computation-communication overlapping improves the overall performance of the application, however, the amount of communication overhead hidden by this technique depends upon the amount of computation available. In our test application, up to 38% of the communication overhead is hidden.

Decomposing data and computation across GPUs can potentially (i) speed up the overall application performance, and (ii) reduce the application memory footprint required per GPU.

In this work, we presented two applications with different data exchange requirements to demonstrate the effectiveness of our computation decomposition approach for: (i) achieving almost linear speedup when no communication overhead is imposed or the communication overhead is completely overlapped with kernel execution, and (ii) partitioning data such that problems with memory footprint larger than the available GPU memory can be solved by GPU systems.

In the linear algebra application, where decomposition does not impose data exchange among GPUs, a linear speedup is achieved; and in the CFD application, no linear speedup is achieved but the memory footprint is distributed among GPUs.

Although the computation performance of the CFD application scales linearly with decomposition, the overall performance does not scale linearly because of the increased communication overhead. This is due to the fact that the PCI Express bandwidth per GPU degrades as the number of GPUs increases.

The increased communication overhead is due to two factors: (i) bandwidth degradation due to concurrent data exchange, and (ii) limited support for data exchange provided by OpenCL.

These two issues, specially the later, should be improved in order to fully exploit the computational capabilities of systems with two or more GPUs. As noted earlier, in the future with better hardware communication mechanisms our approach will deliver much more effective solutions.

CHAPTER 7. Coarse Grain Computation-Communication Overlap for Efficient Application-Level Checkpointing for GPUs

7.1 Introduction

GPU design aims to hide memory latency rather than reducing it as in the case of the CPU. This design philosophy allows for allocating more transistors to computing resources rather than to logic control, resulting in a GPU design that is composed of hundreds of light cores with relatively simple logic control.

However, large computation capabilities does not translate directly into high performance. Even with hundred of cores available, it is possible to have idle computing resources due to memory latencies.

To hide memory latency and make efficient use of all the computing hardware (hundreds of light cores), the GPU generates a large number of threads. Threads may be in any of three states: executing, ready, or waiting. An executing thread is currently running in a GPU core. A ready thread has all the necessary data and it is waiting for available computing resources (GPU cores) to begin its execution. Finally, waiting threads are waiting for the required data to arrive from the host memory. When executing threads stall due to memory latency, the ready threads start execution while the ones stalling wait for the data to be completely transferred into the device memory [83]. This technique, known as overlapping computation-communication, has been largely used for achieving high performance in the presence of large memory latencies [81]

.

7.2 Application-Level Checkpoint for GPUs

Any checkpoint/restart scheme for a GPU accelerated application must consider the challenges imposed by the GPU design:

- *Slow device-to-host data transfer* - GPU-based systems are characterized by slow GPU (device) to CPU (host) data transfers. Therefore, a checkpoint scheme with large data transfers, or high frequency of transactions, will have a significant negative impact on the performance of the application.

- *Synchronization* - Given the large number of in-flight threads, the global synchronization approach commonly found in checkpointing schemes is not desirable. Instead, a new synchronization approach is necessary to decrease delays and to avoid idle times in hardware.

- *Flexibility* - GPGPU programming languages offer great flexibility to the GPU-based systems. A good checkpointing scheme will take advantage of the flexibility that these high level languages provide, and be implemented in such a way as to have minimal interference during execution.

Our scheme addresses these challenges by using a variety of methods.

- To deal with the slow device-to-host data transfer, our checkpointing scheme decomposes the problem (computation). By decomposing the problem in smaller pieces it is possible to start checkpointing the process as soon as possible.

- To minimize computation idling due to large data transfers, we propose overlapping GPU computation with CPU communication. This overlapping takes advantage of problem decomposition approach and the GPU native block communication paradigm.

- Stemming from our problem decomposition approach, and utilizing the overlapping computation and communication, we achieve low overhead by overlapping checkpointing with computation.

7.2.1 Computation Decomposition

A GPU requires that all the data involved in the computation is stored locally before and after each computation. This means that all required data is transferred from memory before the computation, and to memory after the computation. Figure 7.1a illustrates the GPU execution model, where the execution time is

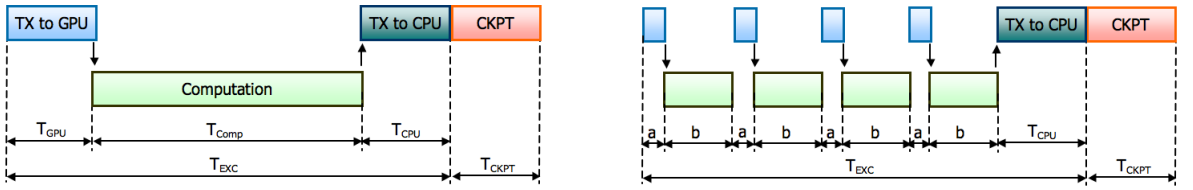
$$T_{EXC} = T_{GPU} + T_{Comp} + T_{CPU} \tag{7.1}$$

and the checkpoint overhead is

$$T_{Overhead} = T_{CKPT} \tag{7.2}$$

In this execution model the GPU waits T_{GPU} before starting the computation. Shown in Figure 7.1b the waiting time is reduced by dividing the whole process into n smaller subprocesses.

As seen in Figure 7.1b, a subprocess before starting its computation should have all the data at the device memory.



(a) Standard Execution Model

(b) Process divided into n subprocesses, where $a = \frac{T_{GPU}}{n}$ and $b = \frac{T_{Comp}}{n}$

Figure 7.1: Execution and Checkpointing approaches on a GPU

7.2.2 Overlapping GPU Computation with CPU Communication

The objective of dividing the process into n subprocesses is to overlap the CPU communication with the GPU computation. Ideally, no computation resource idles if each subprocess computation time is bigger or equal to the transfer time, or

$$\frac{T_{GPU} + T_{CPU}}{n} \leq \frac{T_{Comp}}{n} \tag{7.3}$$

This ideal scenario is shown in Figure 7.2, and the execution time is given by

$$T_{EXC} = \frac{1}{n}(T_{GPU} + T_{CPU}) + T_{Comp} \tag{7.4}$$

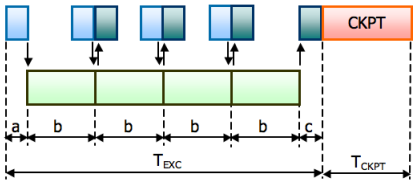


Figure 7.2: GPU Computation-Communication overlap, where $a = \frac{T_{GPU}}{n}$, $b = \frac{T_{Comp}}{n}$ and $c = \frac{T_{CPU}}{n}$

Recall that the GPU's computation time T_{Comp}, depends on the number of generated threads. As introduced earlier, a GPU requires a large number of threads to hide latency and to make efficient use of the computation resources. Therefore, the number of subprocesses n should be chosen in such a way that there is still enough threads on the subprocess to keep all the computation resources busy at any time during the execution.

Theoretically, by optimal dividing the problem into n subprocesses, the execution time is reduced by $T_{reduced}$ that can be achieved to be as given below.

$$T_{reduced} = \frac{n-1}{n}(T_{GPU} + T_{CPU}) \tag{7.5}$$

Where n is the reduction factor. In Equation 7.5 the number of subprocesses, n, is limited by (i) the amount of computation at any subprocess, i.e. the number of threads available, (ii) the subprocess creation overhead.

7.2.3 Overlapping the Checkpointing with GPU Computation

As show in Figure 7.2 checkpoint is possible only after the data is stored in the host memory. In the same way that GPU computation and CPU communication overlaps, it is possible to overlap GPU computation with the checkpointing process, see Figure 7.3. This overlapping is possible because the checkpointing process is executed exclusively at the CPU.

In this scenario the execution time is still given by Equation 7.4. However, the checkpoint overhead is

$$T_{Overhead} = \frac{1}{n}T_{CKPT} \tag{7.6}$$

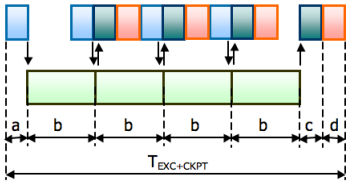


Figure 7.3: GPU Operation overlapped with Checkpointing, where $a = \frac{T_{GPU}}{n}$, $b = \frac{T_{Comp}}{n}$, $c = \frac{T_{CPU}}{n}$ and $d = \frac{T_{CKPT}}{n}$

Using Equations 7.5 and 7.6, the total time saved for execution and checkpointing is

$$T_{Saved} = \frac{n-1}{n}(T_{GPU} + T_{CPU} + T_{CKPT}) \tag{7.7}$$

The constraint defined in inequality defined by Equation 7.3 is enough to guarantee correctness, because (i) the checkpointing process is done at the CPU level using a different bus, therefore does not increase the traffic at the PCIe bus, and (ii) the checkpoint is saved asynchronously to avoid blocking the CPU.

7.3 Results

Good test cases for our checkpointing scheme are applications whose computation is nicely divided into sets of smaller processing jobs. Where each processing job modifies a different set of data. There are several application-types with these characteristics, such as: matrix multiplication, fast fourier transform, etc. Matrix multiplication is a well known scientific benchmark [84, 85, 33], and it can be used to show the results achieved by our application-level checkpointing scheme without loosing generality in our approach.

To illustrate the problem, the matrix multiplication is defined as:

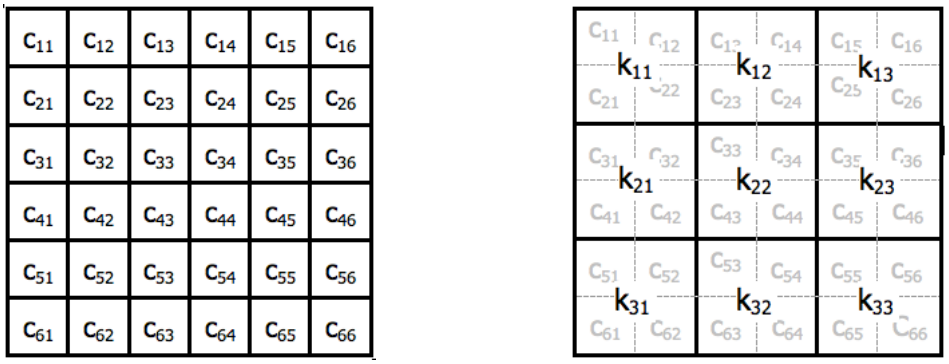
$$C_{m \times p} = A_{(m \times q)} \times B_{(q \times p)} \tag{7.8}$$

Since NVidia SDK provides optimal kernels for square matrix multiplication, the problem reduces to the case where A and B are square matrices, i.e. m = p = q.

This section shows how the computation decomposition approach was implemented for the matrix multiplication problem, and the results obtained from this implementation.

7.3.1 Implementing Computation Decomposition

Figure 7.4 shows the strategy followed to divide the matrix multiplication problem in a set of smaller chunks or subproblems.



(a) A $m \times m$ matrix, $m = 6$ (b) Division into n^2 submatrices, n=3

Figure 7.4: Division strategy for the matrix multiplication

Figure 7.4a shows the resulting matrix C which is divided in Figure 7.4b into n^2 smaller square matrices. This strategy is taken because it optimizes memory transfers, for instance submatrix k_{11} requires only rows one and two and columns one and two from matrices A and B, respectively. To simplify memory transfers we store matrix A row wise and matrix B column wise, which is possible because transposing a matrix in a GPU requires negligible time.

Although the strategy divides the communication and computation into n^2 submatrices, the checkpoint process is divided only into n subprocesses, each composed of all the submatrices in the same row. This is done to optimize the data transfer. Instead of saving several smaller non-contiguous memory areas, only one larger contiguous memory area is saved.

7.3.2 Experimental Results

The proposed scheme is tested on a system composed of an Intel Xeon Quadcore processor and a Nvidia Tesla T10 GPU. The application and the checkpointing code is written using CUDA v.2.3.

Table 7.1 shows the matrix sizes and number of subprocesses utilized in our test cases.

Table 7.1: Parameters used in the test cases of the Matrix Multiplication

Parameter	**Value**
Matrix Size (m)	1,024, 2,048, 4,096, 8,192, 16,384
Checkpoint Subprocesses (n)	1, 2, 4, 8, 16, 32

Next, the execution time and checkpoint overhead for the proposed checkpointing scheme are presented. Notice that for brevity, we only present results for the two extreme cases in Tables 7.2 and 7.3 for the smallest and the biggest matrices in the experiments.

Table 7.2: Matrix Multiplication with m = 1,024

n	**Execution Time (msec)**	**Checkpoint Overhead (msec)**	**Total Time (msec)**
1	16.01	1.98	17.99
2	15.72	1.11	16.83
4	19.67	0.64	20.31
8	22.42	0.33	22.75
16	37.06	0.17	37.23
32	126.19	0.08	126.27

Table 7.3: Matrix Multiplication with m = 16,384

n	**Execution Time (msec)**	**Checkpoint Overhead (msec)**	**Total Time (msec)**
1	47,468.03	1,531.50	48,999.53
2	47,105.95	773.06	47,879.01
4	47,154.36	382.36	47,536.72
8	47,610.95	185.26	47,796.21
16	49,089.46	92.09	49,181.55
32	52,379.07	46.77	52,425.84

It can be seen in Table 7.3 that overlapping achieves better execution time until a point where the execution time starts increasing. This increase starts at $n^2 = 64$ and it is primarily due to the fact that there is not enough threads to keep the computing resources busy. For the case show in Table 7.2 with a matrix size of 1024, it is even more critical as the time starts increasing at $n^2 = 16$, which is reasonable in light of the previous explanation.

Tables 7.2 and 7.3 shows that the relation showed in Equation 7.6 is met. Notice that the checkpointing overhead is approximately halved as n is doubled. The checkpoint overhead decreases as n increases. The value of n is mainly determined by whether or not there is enough computation on the computing resources.

Figure 7.5 shows the normalized overhead execution time introduced by every implementation. It can be noticed that for some number of checkpoint subprocesses, the performance is better than that of the non-overlapped non-checkpointed case.

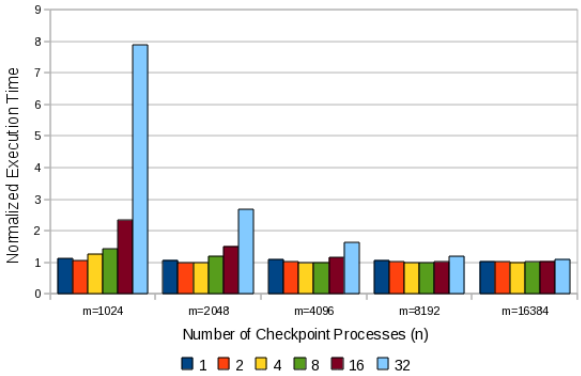


Figure 7.5: Execution time overhead imposed by the different implementations

7.4 Conclusion

We have proposed an approach for application-level checkpointing that provides flexibility and different levels of granularity for checkpointing.

Overlapping GPU computation with checkpointing allows for very low checkpoint overhead with respect to the non-overlapped case. The overhead time for checkpointing can be reduced by a factor of n. However, notice that both theoretically and experimentally, n is limited by the amount of computation available to keep the computing resources busy.

We aim for low performance overhead by implementing the checkpoint process as a multi-threaded process that saves the checkpoint data asynchronously, which releases the CPU to perform other tasks.

Our implementation results show that indeed overlapping operations and checkpointing achieves both, the low checkpoint overhead and reduces execution time.

CHAPTER 8. Data Flow-Based Application-Level Checkpointing for GPU Systems

8.1 Introduction

Like any computing engine of today, reliability is also a concern in GPUs. GPUs are particularly vulnerable to transient and permanent failures. Application execution time is affected negatively by the presence of such failures. A popular fault tolerant scheme is the Checkpoint/Restart technique [14]. This scheme takes periodic snapshots of the application state (checkpoints) during the execution. Upon an occurrence of a failure, the application restarts from the last valid checkpoint.

Application-level checkpointing is more efficient than system-level and user-level checkpointing because of its efficiency in terms of checkpoint overhead. In application-level checkpointing the pieces of code that save the application state, i.e. the checkpointing code, are manually inserted by the application developer, which requires specific knowledge of the application, it is complex and prone to errors.

An alternative to manually inserting checkpointing code is automatic insertion at pre-compile or compile time. Inserting automatically checkpointing code at pre-compile time implies program transformation. Program transformation to provide fault tolerance has been developed in several environments, such as embedded systems [86], MPI systems [87], etc.

In this chapter we describe our scheme for automatic application-level checkpointing of OpenCL applications. This scheme analyzes data dependencies among kernels and data access patterns to transform a non-checkpointing capable OpenCL application into one that supports checkpointing.

8.2 Application-Level Checkpointing

The major goal of the *Application Transformation* stage is to enhance an OpenCL application with constructions that efficiently save the state of the application. Efficient checkpointing implies low checkpoint overhead, which is achieved by reducing the amount of data checkpointed and hiding the checkpoint latency. The pieces of code added to the application perform data transfers from GPU memory to main memory and optimizations to reduce the checkpoint overhead.

This section describes how the information provided by the data analyzer is utilized for program transformation. Data dependencies are utilized to define the application state and where it is more profitable, in terms of checkpoint overhead, to checkpoint it, while data access patterns allows to determine if computation-communication overhead can be implemented to reduce the checkpoint overhead.

8.2.1 Finding the Application State

The first step towards supporting application-level checkpointing is to identify the data that are part of the state of the application.

The data that compose the state of the application at a given time can be identified by analyzing the data dependencies. The data dependence graph provides information regarding the inputs required and outputs generated by kernels, as well as data dependencies among kernels, which allows to define the state of the application for each kernel.

For an OpenCL application composed of N_K kernels, there are potentially N_K different application states producing different checkpoint overheads. The application state for a snapshot taken in between two contiguous kernels $\mathrm{K_i}$ and $\mathrm{K_j}$ executed sequentially is defined by the set of inputs (I) required by the kernel $\mathrm{K_j}$ or the set of outputs (O) generated by the kernel $\mathrm{K_i}$, and the set of data required by other kernels, i.e. future inputs, (F), see Equation 8.1.

$$App.\ State_i = \underbrace{\bigcup_k O_{i,k} + \bigcup_k F_{i,k}}_{\text{After kernel } K_i} = \underbrace{\bigcup_k I_{j,k} + \bigcup_k F_{j,k}}_{\text{Before kernel } K_j} \tag{8.1}$$

Algorithm 5 presents the main steps for building the application state for a kernel $\mathrm{K_i}$, it is

important to notice that future inputs can be originated by inputs to kernels that are executed before or after K_i, lines 5 to 7 and lines 8 to 10, respectively.

Algorithm 5 Finding the Application State of the kernel K_i

```
1: Add all output parameters of K_i to AppState_i
2: {This section finds the Future Inputs}
3: for all kernels K_j in the application with i ≠ j do
4:    for all input parameters P_k of K_j that depends on an output of other kernel K_r do
5:       if kernel K_j is executed before K_i and K_r is executed between K_j and K_i then
6:          Add parameter P_k to AppState_i
7:       end if
8:       if kernel K_j is executed after K_i and K_r is executed before K_i or after K_j then
9:          Add parameter P_k to AppState_i
10:      end if
11:   end for
12: end for
```

As an example, Table 8.1 shows the inputs, outputs and future inputs for each kernel in the dependence graph of Figure 5.3.

Table 8.1: Inputs, Outputs, Future Inputs and the Application State for the data dependence graph defined in Figure 5.3

Kernel	Input	Output	Future	App. State
K_0	b[·]	d[·]	b[·], c[·]	b[·], c[·], d[·]
K_1	b[·], c[·]	a[·]	b[·], d[·]	a[·], b[·], d[·]
K_2	a[·], b[·]	a[·]	d[·]	a[·], d[·]
K_3	a[·], d[·]	b[·]	-	b[·]
K_4	b[·]	c[·]	b[·]	b[·], c[·]

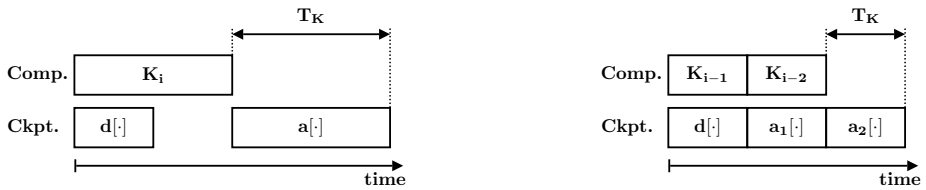
From now on, the application state of a kernel is referred in terms of its outpus and future inputs, hence, the time for transferring the application state, i.e. the checkpoint latency, is defined in Equation 8.2.

$$T_{App.\ State_i} = \underbrace{\sum_{O_{i,j} \in \Omega_i} T_{O_{i,j}} + \sum_{O_{i,j} \notin \Omega_i} T_{O_{i,j}}}_{T_{Output_i}} + \underbrace{\sum_{j} T_{F_{i,j}}}_{T_{Future_i}} \tag{8.2}$$

where: $\Omega_i = \{O_{i,j} | O_{i,j}$ has linear access$\}$

8.2.2 Selecting a Checkpoint Location

Finding where to checkpoint is approached as choosing an application state, among the ones previously found, that achieves the lowest checkpoint overhead. Low checkpoint overhead is achieved by checkpointing a small application state and overlapping checkpointing with kernel computation. Overlapping checkpointing with kernel computation requires linear access on the kernel outputs in order to decompose kernel computation and checkpointing, otherwise, the amount of overlapping achieved is limited. Figure 8.1 shows the effect of linear and non-linear output access on the checkpoint overhead (T_K).



(a) No linear access on output

(b) Linear access on output

Figure 8.1: Overlapping checkpointing with kernel computation

Next, two schemes for choosing the application state are presented: (i) with no runtime information, and (ii) with runtime information, i.e. array sizes and kernel execution times.

If no runtime information is provided, choosing the application state that achieves the lowest checkpoint overhead takes only into consideration the application states previously found. Towards this objective, only the states that cannot be expressed in terms of other states are considered, see Equation 8.3.

$$App.\ State_i \not\supset App.\ State_j,\ \ \forall\, j < N_K,\ i \neq j \tag{8.3}$$

For instance in the example shown in Table 8.1 the application states of kernels K_0, K_1 and K_4 can be expressed in terms of the application state of K_3, hence, the potential application states for checkpointing are the ones for K_2 and K_3. Among those two potential locations for checkpointing the application, the one with less arrays is picked as the checkpoint location, which in this case is K_3.

While choosing an application state based solely on data dependences and number of arrays that compose the application state is simple, it might not be the best solution because there is no guarantee that the application state has indeed the smallest checkpoint size or no overlapping is possible due to a kernel with small execution time.

Runtime information allows to choose a location for checkpointing where a balance between the checkpoint size and the kernel execution time attains low checkpoint overhead. Runtime information is provided as part of the XML application structure file, and it can be absolute, as shown in Listing 8.1, or relative, using percentages or fractions.

```
1  <runtime>
2    <kernel>
3      <name>K1</name>
4      <time>1,009.15msec</time>
5    </kernel>
6    <array>
7      <name>a</name>
8      <size>128KB</size>
9    </array>
10 </runtime>
```

Listing 8.1: A XML application structure file with runtime information

Two metrics are identified to select an application state using runtime information: the checkpoint ratio and the minimum checkpoint overhead, see Equations 8.4 and 8.5. The checkpoint ratio provides a measure of the amount of checkpoint overhead that can be overlapped by kernel execution, hence, the bigger this ratio the better. The minimum checkpoint overhead provides a measure of the amount of checkpoint overhead that cannot be overlapped by kernel execution, hence, the shorter the better. The factors for computing the checkpoint ratio can be expressed as absolute or relative values, while the factors for computing the checkpoint overhead can be expressed only as absolute values, i.e. time.

$$Ckpt.\ Ratio_i = \frac{Kernel\ Computation_i}{\sum_j T_{F_{i,j}} + \sum_{O_{i,j} \in \Omega_i} T_{O_{i,j}}} \tag{8.4}$$

$$(T_K^{min})_i = max\big(0, Overlapped_i\big) + \sum_{O_{i,j} \notin \Omega_i} T_{O_{i,j}} \tag{8.5}$$

$$\text{where: } Overlapped_i = \sum_j T_{F_{i,j}} + \sum_{O_{i,j} \in \Omega_i} T_{O_{i,j}} - T_{Kernel_i}$$

Although the minimum checkpoint overhead metric provides direct information for selecting the location that achieves the lowest checkpoint overhead, it can be computed only if absolute runtime information is provided. Therefore, providing absolute information results in a more accurate selection of the checkpoint location.

To illustrate the usage of relative runtime information, let us assume in the example of Table 8.1 that the execution time percentage for K_1 and K_2 is 35% and for K_0, K_3 and K_4 is 10%, all the output data access patterns are linear and the array sizes for a and c is x *bytes* and for b and d is $x/2$ *bytes*. For brevity, only the two highest normalized checkpoint ratios are presented, which correspond to K_2 and K_3. The normalized checkpoint ratios for K_2 and K_3, 1.0 and 0.86 respectively, suggest that the best location for checkpointing is K_2, which is different than the location selected with no runtime information.

Next, to compare the results obtained using relative information with absolute information, let us assume that the execution time percentage for K_1 and K_2 is 350 msec and for K_0, K_3 and K_4 is 100 msec, and the transter time for arrays a and c is 300 msec and for arrays b and d is 150 msec. With this information, the minimum checkpoint overhead for K_2 is 100 msec and for K_3 is 50 msec, which suggest that the best location for checkpointing is K_3. Since the minimum checkpoint overhead metric provides more accurate information, K_3 is selected as the checkpoint location.

8.2.3 Kernel Decomposition

Once the location for checkpointing have been defined and provided that the kernel output access is linear, the kernel computation and hence the output data should be decomposed to allow overlapping kernel execution with application checkpointing.

Kernel decomposition is performed by partitioning the grid of work-groups into smaller grids as shown in Figure 8.2. Decomposing the kernel computation might have a negative effect on the overall performance, because as the number of partitions increases, each new kernel partition might not have enough computation to keep the hardware busy, hence, the total execution time increases, i.e. imposes partition overhead. Hence, a kernel should be decomposed into a number of partitions that keeps at the same time low checkpoint overhead

and partition overhead [88].

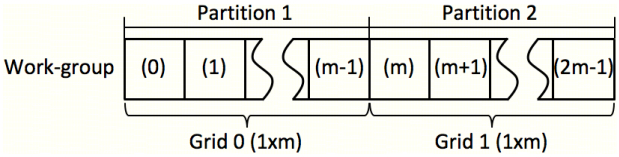
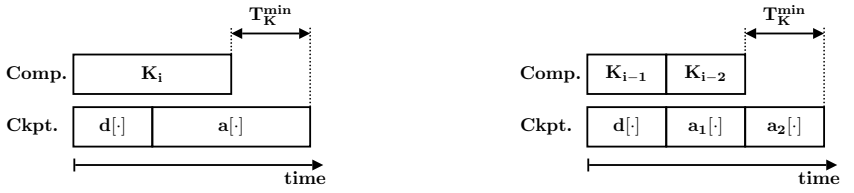


Figure 8.2: Kernel computation decomposition into two grids

Although the effect of kernel decomposition depends on certain inherent features of each application, next, a reference value for the number of partitions is provided. When the checkpoint latency is bigger than the kernel execution time and the application state is composed of future inputs, the number of partitions (M) in which a kernel is decomposed is given in Equation 8.6.

$$M = max\left(\left\lceil \frac{T_{Kernel_i}}{T_{Future_i}} \right\rceil, \left\lceil \frac{T_{Output_i}}{(T_K^{min})_i} \right\rceil\right) \tag{8.6}$$

The intuitions behind Equation 8.6 are: (i) the overlapping technique cannot hide more checkpoint latency than what it is possible when data is transferred continuously as in Figure 8.3a, hence, the kernel is decomposed in fractions equal to the size of either the minimum checkpoint overhead (T_K^{min}) or the future data (T_{Future}), and (ii) the size of future data limits the amount of checkpoint latency that can be overlapped, see Figure 8.3b, hence, increasing the number of kernel partitions beyond the optimal number does not return any additional gain.



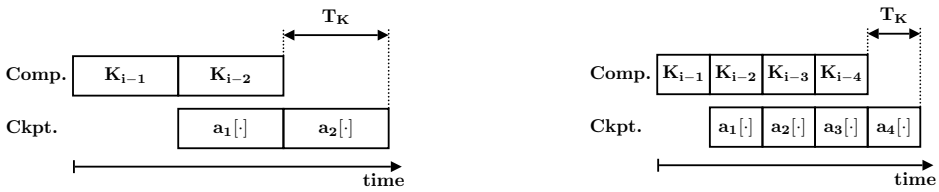
(a) Maximum overlap achievable

(b) Maximum overlap achieved through efficient decomposition

Figure 8.3: Decomposition when the application state is composed of output and future data and it is larger than kernel computation

When the checkpoint latency is smaller than the kernel execution time or the application is composed only of output data, there is no limit for the number of partitions in terms of achieving

the best overlapping possible. Figure 8.4 shows that increasing the number of partitions, the checkpoint overhead decreases by a factor of M.



(a) Two Partitions (b) Four parttions

Figure 8.4: Decomposition when the application state is composed only of output data or it is shorter than kernel computation

The number of partitions can be increased up to a point where the total execution time starts to increase because no enough computation is available on the kernel partitions to hide long latency operations. In this context, the limit for the number of partitions is given by the inherent features of the application, i.e type of computation, memory access, problem size, etc. In this case, the overhead imposed by data decomposition, i.e. D_{ovh}, is utilized for computing a reference value for the number of data partitions and, hence, kernel partitions, see Equation 8.7. In general the total data decomposition overhead should be kept within a percentage, i.e. $D_{\%}$, of the total output time.

$$M \cdot D_{ovh} \leq D_{\%} \cdot T_{Output_i} \implies M \leq D_{\%} \cdot \frac{T_{Output_i}}{D_{ovh}} \tag{8.7}$$

8.3 Adding Application Support for Application-Level Checkpointing

Transforming an application for supporting application-level checkpointing is achieved by:

- Providing support for concurrent checkpoint operations and kernel execution through OpenCL structures,
- Decomposing kernels and generating offsets, and
- Adding checkpoint operations.

An OpenCL context includes devices, objects for kernels and memory, and command queues. The host program places commands in a command queue, which are scheduled for execution

on the device that is associated with the queue. Concurrent memory operations and kernel execution is supported through separate command queues for memory and kernel execution commands.

Providing support for concurrent checkpoint, i.e. memory, operations and kernel execution requires to define and associate an additional command queue to the GPU, which is used for receiving and scheduling checkpoint operations, as well as to redefine the kernel object as an array to handle kernel partitions. In addition to those structures, synchronization events are needed since kernel execution and checkpoint operations are executed in different queues, see Figure 8.5.

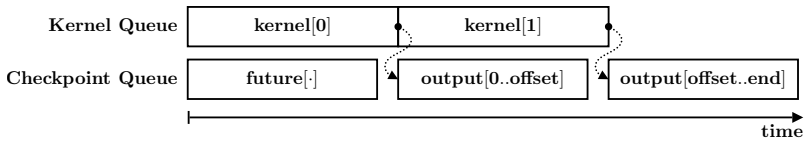


Figure 8.5: Synchronization among kernel execution and checkpoint operations

As described in Section 8.2.3, kernel decomposition is achieved by partitioning the grid of work-groups into smaller grids. This decomposition modifies the work-item global and work-group identifiers, and therefore, it modifies the data access, because, in general the indexes used to access data are based on those identifiers. Hence, the kernel code needs to be transformed to support decomposition, which involves adding a new parameter to the declaration that receives the offset, and modifying the array indexes using the offset.

At the host code level kernel decomposition is supported by computing offsets and sizes of data and kernel partitions and passing parameters to every kernel partition. The kernel partition size is described next in Equation 8.8.

$$\begin{aligned} size(kernel_i) &= \left\lceil \frac{size(kernel)}{M} \right\rceil \\ size(kernel_M) &\leq size(kernel_i), \quad 1 \leq i < M \end{aligned} \tag{8.8}$$

The offset is presented in Equation 8.9, which is based on the kernel partition number and on the kernel partition size.

$$offset_i = (i-1) \cdot \left\lceil \frac{size(kernel)}{M} \right\rceil \tag{8.9}$$

The data partition size is computed as in Equation 8.10.

$$elements(array_i) = \frac{size(kernel_i)}{size(kernel)} \cdot elements(array) \tag{8.10}$$

Checkpoint support is provided by enhancing the host code with memory commands, associated to checkpoint command queue, to transfer the application state to the CPU memory, as well as to flush commands. Next, Listing 8.2 summarizes the checkpoint operations required at host code level.

```
0  cl_kernel kernel[MAX_KERNELS];
1  // ... set kernel parameters for kernel[0] and kernel[1]
2  // ...
3  // overlaps kernel execution and checkpoint operations
4  clEnqueueNDRangeKernel(krnl_queue, kernel[0], 1, NULL, PartSize, WGSize, &events[0] );
5  clEnqueueReadBuffer(ckpt_queue, future, CL_FALSE, 0, size , hFuture, 0, NULL, NULL);
6  clFlush(krnl_queue);
7  clFlush(ckpt_queue);
8  clEnqueueNDRangeKernel(krnl_queue, kernel[1], 1, NULL, PartSize, WGSize, &events[1] );
9  clEnqueueReadBuffer(ckpt_queue, output, CL_FALSE, 0, offset, hOutput, 1, &events[0], NULL);
10 clFlush(krnl_queue);
11 clFlush(ckpt_queue);
12 clEnqueueReadBuffer(ckpt_queue, output, CL_FALSE, offset, offset, hOutput + offset, 1, &events[1],
       ➥ NULL);
13 clFlush(ckpt_queue);
```

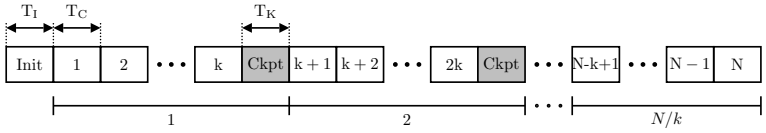
Listing 8.2: Host code for overlapping kernel execution and checkpoint operations

8.4 The Checkpoint Interval

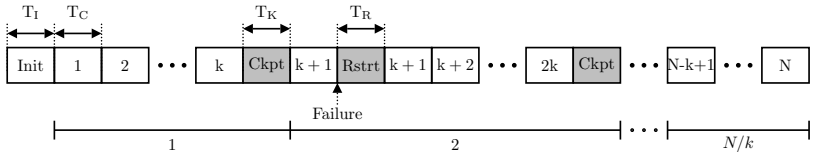
The checkpoint approach takes periodic snapshots of the application state and saves them into stable storage, Figure 8.6a; upon a failure occurrence, the application is restarted from the most recent checkpoint, Figure 8.6b. An efficient checkpointing scheme aims to minimize the total checkpoint overhead imposed, which is achieved by selecting a checkpoint location where the checkpoint size is small and by choosing an appropriate checkpoint interval.

The scheme for selecting a location for checkpointing was introduced in Section 8.2.2. This section focuses on the selection of the checkpoint interval.

For an application with a set of kernels running repeatedly N times inside a loop and a initialization time T_I, the total execution time and checkpoint interval are defined in Equation 8.11 and Equation 8.12. Equation 8.12 relates selection of the checkpoint interval to the number of loop iterations between two successive checkpoints and the time for each iteration,



(a) The checkpoint technique for an application with N iterations



(b) Restarting after a failure using the most recent checkpoint

Figure 8.6: The checkpoint/restart technique

i.e. k and T_C, respectively.

$$T_T = N \cdot T_C + T_I \tag{8.11}$$

$$T_{CI} = k \cdot T_C \tag{8.12}$$

Similarly to other works, in this work the failures in a computational system are modeled as a Poisson process with the failure rate exponentially distributed [89, 90, 91]. The probability of occurring and not occurring a failure in an interval Δt is given by Equations 8.13 and 8.14, respectively.

$$P(t \leq \Delta t) = 1 - e^{-\Delta t \cdot \lambda} \tag{8.13}$$

$$P(t > \Delta t) = e^{-\Delta t \cdot \lambda} \tag{8.14}$$

In this work we propose to choose the optimal checkpoint interval that guarantees that the penalty imposed by a failure in an implementation supporting checkpointing is less than that of an implementation with no checkpoint support, i.e. $T^{(p)}_{Non-Ckptd} \geq T^{(p)}_{Ckptd}$.

The failure interval (T_f) in a checkpointed application is defined as the interval where a failure can occur, which includes the checkpoint interval and the checkpoint overhead as shown below in Equation 8.15.

$$T_f = T_{CI} + T_K = k \cdot T_C + T_K \tag{8.15}$$

8.4.1 One Failure per Execution

The penalty imposed by a failure in implementations with and without checkpoint support are shown in Equations 8.16a and 8.16b, respectively. Equation 8.16a introduces the probability of failure per iteration, i.e. P_f to estimate the average penalty. The penalty paid by a non-checkpointed implementation is computed based on the probability of failure at iteration i, the amount of work lost at that iteration and the application initialization time (T_I); whereas the penalty paid by a checkpointed implementation is computed based on the time used for checkpointing, the average amount of work lost due to a failure and the restart time (T_R).

$$T^{(p)}_{Non-Ckptd} = \sum_{i=1}^{N} (1 - P_f)^{i-1} \cdot P_f \cdot (i \cdot T_C + T_I) \tag{8.16a}$$

$$T^{(p)}_{Ckptd} = \left(\frac{N}{k} - 1\right) \cdot T_K + \frac{1}{2} \cdot (k \cdot T_C + T_K) + T_R \tag{8.16b}$$

By using the summation of geometric series and the Taylor expansion equations, the inequality $T^{(p)}_{Non-Ckptd} \geq T^{(p)}_{Ckptd}$ is simplified in Equation 8.17.

$$P_f \cdot N \cdot T_T \geq \left(\frac{N}{k} - 1\right) \cdot T_K + \frac{k \cdot T_C + T_K}{2} + T_R \tag{8.17}$$

Which leads to the quadratic expression:

$$-\frac{1}{2} \cdot k^2 + \left[\frac{P_f \cdot N \cdot T_T - T_R}{T_C} + \frac{T_K}{2 \cdot T_C}\right] \cdot k - N \cdot \frac{T_K}{T_C} \geq 0 \tag{8.18}$$

According to the quadratic expression of Equation 8.18, the optimal solution for k lies in the interval:

$$N \cdot P_f \cdot \left(N + \frac{T_I}{T_C}\right) + \frac{T_K}{2 \cdot T_C} - \frac{T_R}{T_C} \pm \sqrt{\left[N \cdot P_f \cdot \left(N + \frac{T_I}{T_C}\right) + \frac{T_K}{2 \cdot T_C} - \frac{T_R}{T_C}\right]^2 - 2 \cdot N \cdot \frac{T_K}{T_C}} \tag{8.19}$$

The interval exists if the discriminant inside the square root of Equation 8.19 is positive. Assuming at most a single failure during the application execution, i.e. $P_f \cdot N \approx 1$, the condition for a positive discriminant is shown in Equation 8.20, which always holds.

$$N \cdot T_C > T_K \tag{8.20}$$

The optimal value for k is obtained by differentiating Equation 8.16b with respect to k and equating to zero.

$$k_{opt} = \left\lfloor \sqrt{2 \cdot N \cdot \frac{T_K}{T_C}} \right\rfloor \tag{8.21}$$

By plugging in the obtained k_{opt} into Equation 8.18, it is straight forward to prove that this optimal value falls inside the range of values defined in Equation 8.19

8.4.2 One or More Failures per Execution

To estimate the checkpoint interval based on the penalty associated with more than one fault, the average number of total and failed attempts needed to complete N calculations of duration Δt are given first in Equations 8.22a and 8.22b.

$$attempts = \frac{N}{P(t > \Delta t)} = N \cdot e^{\Delta t \cdot \lambda} \tag{8.22a}$$

$$f(\Delta t) = \frac{N}{P(t > \Delta t)} - N = N(e^{\Delta t \cdot \lambda} - 1) \tag{8.22b}$$

Now, the penalty imposed by faults for the checkpointed and non-checkpointed versions are presented in Equations 8.23a and 8.23b. The penalty paid by the non-checkpointed implementation is computed based on the number of failed attempts to complete the N iterations, the average amount of work lost and the initialization time (T_I); whereas the penalty paid by the checkpointed implementation is computed based on the checkpoint time, the number of failed attempts to complete a failure interval, the average amount of work lost and the restart time (T_R).

$$T^{(p)}_{Non-Ckptd} = f\big(N \cdot T_C\big) \cdot \left(\frac{N \cdot T_C}{2} + T_I\right) \tag{8.23a}$$

$$T^{(p)}_{Ckptd} = \left(\frac{N}{k} - 1\right) \cdot T_K + f\big(T_f\big) \cdot \left(\frac{1}{2} \cdot T_f + T_R\right) \tag{8.23b}$$

By using Equation 8.22b and the Taylor expansion, the inequality $T^{(p)}_{Non-Ckptd} \geq T^{(p)}_{Ckptd}$ is simplified in Equation 8.24.

$$N \cdot T_C \cdot \lambda \cdot \left(\frac{1}{2} \cdot N \cdot T_C + T_I\right) \geq \left(\frac{N}{k} - 1\right) \cdot T_K + \frac{N}{k} \cdot T_f \cdot \lambda \cdot \left(\frac{1}{2} \cdot T_f + T_R\right) \tag{8.24}$$

Which leads to the quadratic expression:

$$-k^2 - \frac{2}{Tc} \cdot \left[T_K - \frac{T_K}{\lambda \cdot N \cdot T_C} - \frac{N}{2} \cdot T_C - T_I + T_R \right] \cdot k - \frac{2}{Tc} \cdot \left[\frac{1}{2} \cdot T_K + T_R + \frac{1}{\lambda} \right] \cdot \frac{T_K}{T_C} \geq 0 \tag{8.25}$$

Solving Equation 8.25 provides a range for values for the parameter k for which a checkpointed application imposes less overhead than a non-checkpointed application.

Similarly to the case with one failure, the optimal solution shown in Equation 8.26 is obtained by differenciating Equation 8.23b with respect to k and equating to zero.

$$k_{opt} = \left\lfloor \frac{T_K}{T_C} \cdot \sqrt{\frac{2}{T_K} \cdot \left(\frac{1}{\lambda} + T_R \right) + 1} \right\rfloor \tag{8.26}$$

8.5 Experimental Results

In this section we present the results for the checkpointing of an OpenCL GPU application utilizing the framework and tool described in this work.

The application used is a CFD application that solves the Navier-Stokes equations on unstructured grids utilizing a high order correction procedure via reconstruction methods. We omit the details of the method used for the application, for further details refer to the paper by Wang [77]. Implementation of this method on GPU systems has been studied in our previous works [36, 92]. This application is composed of five kernels that update local initial values (K_0), neighbor analysis (K_1), cell analysis (K_2), local variables update (K_3) and boundary conditions update (K_4). From the analysis in previous chapters, the data dependencies are presented in Figure 5.3. The single-GPU version of this application is provided as input to the tool for transforming into one that supports application-level checkpointing.

The GPU system utilized for testing and benchmarking is composed of four GPUs NVidia Tesla C2070 with 448 cores and 6GB of global memory, and an Intel Xeon 2.5GHz six-core processor. This system presents a data decomposition overhead (D_{ovh}) experimentally measured of 5 μsec.

The results for the CFD application are presented in Table 8.2, which correspond to a grid with 447,944 cells.

Table 8.2: Kernel and application state timings for the CFD application

Kernel	T_{Kernel} (msec)	T_{Output} (msec)	T_{Future} (msec)	T_K^{min} (msec)	M	T_K (msec)
K_0	4.54	21.31	21.68	38.45	1	-
K_1	49.50	22.36	42.97	15.83	2	-
K_2	34.18	22.36	21.31	9.49	3	13.57
K_3	9.34	21.66	-	12.32	216	-
K_4	0.02	0.02	21.66	21.66	1	-

In Table 8.2 the minimum checkpoint overhead metric (T_K^{min}) is computed according to Equation 8.5, which helps to identify K_2 as the checkpoint location that imposes the least checkpoint overhead. Although not shown in Table 8.2, the checkpoint ratio metric presented in Equation 8.4 selects K_2 as the checkpoint location as well.

Although only the number of partitions (M) for kernel K_2 is needed for implementation purposes, the number of partitions for the other kernels are presented as well for completion purposes. For kernel K_3, the number of partitions was computed to impose a data decomposition overhead ($D_{\%}$) of about 5% of the output time. It is important to notice that for kernels K_0 and K_4 overlaping kernel execution and checkpointing does not help to reduce the checkpoint overhead, that is why the number of partitions is 1.

It is interesting to notice in Table 8.2 that data and kernel decomposition allows to overlap approximately 78% of the checkpoint latency, which represents runtime savings of about 35% per iteration.

The checkpoint overhead presented in Table 8.2, 13.57 msec, represents an increase of approximately 14% per iteration, and exceeds by 4.08 msec to the minimum expected, which is due to the wall timer (1.3 msec), kernel decomposition (175 μsec) and the remaining due to data decomposition and synchronization.

The optimal value for the checkpoint interval (k) is computed using Equation 8.21 and the results in Table 8.2; for a iteration time (T_C) of 97.58 msec, a checkpoint overhead (T_K) of 13.57 msec and 2,000 iterations, the optimal value for k is 23, which translates into 86 checkpoints taken. The total checkpoint overhead incurred by using this k represents approximately

a 4.3% of the checkpoint overhead incurred if a checkpoint is taken at every iteration and an increase of the 0.6% of the total runtime.

8.6 Conclusions

Data dependence analysis is used to identify the application state at different locations, which potentially can be selected as the checkpoint location. The checkpoint location is selected such that the overhead imposed is minimized. Two approaches are identified to select the application state that minimizes the checkpoint overhead: (i) without runtime information, and (ii) with runtime information, the later is more precise than the former, however, in some situations the former can lead to the same result.

Overlapping kernel execution and checkpoint operations is utilized to hide checkpoint latency. The overlapping technique requires to decompose kernel execution, and hence data. We proposed a scheme to calculate the number of kernel partitions aiming to achieve the minimum checkpoint overhead possible; the future data part of the application state limits the minimum checkpoints achievable and, hence, the number of partitions. In our experiments, about 78% of the checkpoint latency is hidden by the overlapping technique.

We presented two approaches for computing the checkpoint interval in terms of number of iterations. These methods differ in the fact that the first method assumes one fault per run, hence, provides an optimal checkpoint interval based on the total runtime, whereas the second method does not have an assumption on the number of failures. By utilizing the first method to compute the checkpoint interval, the checkpoint overhead introduced is reduced by about 96% compared to the case where one checkpoint per iteration is taken.

CHAPTER 9. Conclusion and Future Work

9.1 Conclusion

This research addressed the issues of exploiting higher level of parallelism and developing support for application-level fault tolerance in applications using multiple GPUs. The proposed solution is a framework that (i) analyzes the host and kernel code to build a data dependence graph and to identify data access patterns, (ii) performs kernel and data decomposition, (iii) minimizes the checkpoint overhead, and (iv) transforms the application to support the desired feature.

9.1.1 Data Dependencies and Access Patterns Analysis

This research introduced a framework to enhance single-GPU applications through program transformation. At the heart of this framework is the data analyzer unit. This unit analyzes the data flow among kernels to generate a data dependence graph and the data access patterns of each array stored in the GPU memory to classify the access patterns as linear and non-linear (random) access. The data dependence graph represents kernels as nodes and dependencies as directed edges among the kernels.

The data dependencies and access patterns identified are extremely important for the application transformation, because they (i) define the data organization, i.e. decomposed or replicated across GPUs, the data transfer requirements among GPUs and the computation decomposition, and (ii) define the application state, the checkpoint location and the ability of overlap kernel execution with checkpoint operations.

To simplify the design of the data analyzer unit, as well as the design of the framework, and to allow for providing optional runtime information, we introduced an *application structure*

information file that represents information, such as kernel and runtime information, using XML language.

9.1.2 Kernel (Computation) and Data Decomposition

Kernel decomposition is the building block that allows to exploit data parallelism and reduce checkpoint overhead. This research exploited the limited synchronization among threads for achieving computation decomposition, i.e. no synchronization among threads that belong to different thread groups is supported. The thread blocks were grouped into sets and executed separately and independent from each other. Depending on the desired feature, i.e. multi-GPU execution or application-level checkpointing, these sets of thread groups are executed in parallel in different GPUs or sequentially in the same GPU overlapped with data transfers to reduce checkpoint overhead.

Multi-GPU execution require local copies of the data in the memory of each GPU. In this context, the data dependencies and access patterns identified define the data organization, i.e. decomposed or replicated across GPUs, and the transfer requirements among GPUs.

Since kernel decomposition modifies the global and group identifiers, which are used for accessing data stored in the GPU memory, we utilized data offsets to guarantee proper data access. These offsets are computed based on the number of kernel partitions and passed as parameters to the kernel.

9.1.3 Minimizing the Checkpoint Overhead

The framework developed approached the issue of minimizing the checkpoint overhead in two steps: first, it finds a location in the application logic among a set of potential locations where the checkpoint overhead is potentially the lowest, i.e. the checkpoint location, and second, it calculates an optimal checkpoint interval.

We developed an algorithm for finding the application state at every kernel, which are potential locations for taking checkpoints. To find the checkpoint location, three metrics are defined: (i) the number of arrays to checkpoint, when no runtime information available, (ii) the checkpoint ratio, when relative runtime information is available, and (iii) the minimum

checkpoint overhead, when absolute runtime information is available. We represent runtime information using the XML language as part of the application structure file.

In an iterative application, the checkpoint interval is defined in terms of loop iterations. We proposed an optimal checkpoint interval for two scenarios: (i) assuming at most one failure per run, and (ii) with no assumptions about the number of failures per run.

To further minimize the checkpoint overhead, kernel execution and checkpoint operations are overlapped, which requires computation decomposition. In this context, we analyzed the effect of the number of partitions on computation decomposition, and it was shown that up to a number of partitions the decomposition overhead is negligible; beyond that number of partitions the overhead is noticeable and it is due to the lack of thread blocks, and hence warps, to hide memory and pipeline latency. We proposed expressions for calculating the number of partitions based on the size of the application state and the decomposition overhead.

9.1.4 Automated Application Transformation

The last component of our framework performs automatic program transformation to reduce the burden of implementing the techniques required to support the desired feature. In this direction, the kernel and host code are modified to support data and kernel decomposition. Moreover, multi-GPU execution requires support for multiple devices at the host code level, which includes defining and associating separate command queues to every device supported, redefining kernel and memory objects as arrays, and passing parameters to each kernel partition. We avoid race conditions on data transfers by synchronizing events through OpenCL events. Application-level checkpoint requires support for copying the application state to host memory. Finally, overlapping computation and communication operations separate queues are implemented.

9.2 Future Work

The research addressed the issues of exploiting higher levels of parallelism and application-level checkpoint for single-GPU applications. A framework for enhancing single-GPU application was developed, however, some issues remain open.

9.2.1 Integrating Application-Level Checkpoint and Multi-GPU Execution

At the current state of the design and implementation of our framework to enhance single-GPU applications, in a multi-GPU system a checkpoint can be taken to save the application state of every GPU, however, no coordination among them is implemented. This lack of coordination might result in an increased checkpoint if common data is checkpointed, and, even more important, no consistency in the global application state is guaranteed. Therefore, the natural next step in this research is to implement coordination among the checkpoints taken to guarantee a consistent application state in the checkpoint taken.

9.2.2 Multi-Dimensional Kernels

Currently, our framework supports access pattern analysis and kernel decomposition of only one dimensional arrangements of thread blocks and threads. In order to support higher dimensional arrangements, the access pattern analysis has to consider multiple thread and thread block identifiers. Also, the kernel decomposition needs to consider data organization to avoid unnecessary communication overhead.

9.2.3 Incorporating Direct Transfers Capabilities into the Framework Design

The PCIe interconnection network in combination with the inability to perform direct data transfers between GPUs has proven to be a limiting factor for achieving higher speedups in multi-GPU systems. PCIe bandwidth degrades considerably for concurrent communications.

GPUDirect, a technique supported by Nvidia, exchanges data directly among GPUs without copying it first to a buffer in host memory. In this context, concurrent transfers do not degrade considerably the bandwidth because the PCIe interconnection network implements separate links for concurrent communication, and hence the communication overhead does not increase as it does with transfers using a buffer in host memory. Unfortunately, GPU direct works only with GPUs connected to the same IOH. In this context, achieving higher speedups in systems where the GPUs are connected to the same IOH might only require to perform direct transfers among GPUs instead of copying data to host memory first. However, for systems where the

GPUs are connected to different IOHs data transfers still need to copy the information to a host buffer, hence, the data, and perhaps kernel, decomposition and transfer should be designed considering this constraint.

In particular, in the 4-GPU system utilized for our experiments the GPUs are connected to two different IOHs, hence, a two GPU implementation will benefit from this direct transfer feature; on the other hand, a four GPU implementation still requires copying the information to the host memory.

OpenCL is a platform independent programming language specification, however, the support for the different platforms/architectures is provided by the hardware manufacturers through libraries. Features such as GPUDirect should be supported by a Nvidia library, however, at the time of this writing there is no support for GPUDirect in OpenCL.

9.2.4 Incorporating Data Information to Reduce Communication Overhead

Relying on fastest links is not the only way to reduce the communication overhead, reducing the amount of data exchanged between GPUs will reduce the communication overhead. In this work, the analysis is done at pre-compile time, where only data dependence and access pattern information can be extracted from the application. In this context, a non-linear access potentially imposes data exchange among GPUs with no further analysis. By providing information about the application data: (i) a non-linear access pattern might be transformed into a linear access pattern, or (ii) only required data might be transferred.

BIBLIOGRAPHY

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[2] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, pp. 473–530, Sept. 1982.

[3] G. Moore, "Progress in Digital Integrated Electronics," in *International Electron Devices Meeting*, vol. 21, pp. 11 – 13, 1975.

[4] P. Gelsinger, "Microprocessors for the New Millennium: Challenges, Opportunities, and New Frontiers," in *Solid-State Circuits Conference, 2001. Digest of Technical Papers. ISSCC. 2001 IEEE International*, pp. 22 –25, 2001.

[5] J. Duffy, *Concurrent Programming on Windows*. Addison-Wesley Professional, 1st ed., 2008.

[6] R. Wilson and D. Lammers, *Grove Calls Leakage Chip Designers' Top Problem*. EE Times, Dec. 2002. Available: `http://www.eetimes.com/story/OEG20021213S0040`.

[7] Intel, *Intel director explains why multicore processors offer better performance on less power*. TechRepublic, Mar. 2009. Available: `http://blogs.techrepublic.com.com/itdojo/?p=605`.

[8] Intel, *Intel Multi-Core Technology*. `http://www.intel.com/multi-core/`.

[9] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee:

A Many-Core x86 Architecture for Visual Computing," *ACM Transaction on Graphics*, vol. 27, pp. 18:1–18:15, Aug. 2008.

[10] D. Blythe, "Rise of the Graphics Processor," *Proceedings of the IEEE*, vol. 96, pp. 761 –778, may 2008.

[11] J. Nickolls and W. Dally, "The GPU Computing Era," *Micro, IEEE*, vol. 30, pp. 56 –69, march-april 2010.

[12] I. S. Haque and V. S. Pande, "Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU," in *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 691 –696, may 2010.

[13] E. Roman, *A Survey of Checkpoint/Restart Implementations.* Technical Report LBNL-54942, 2002.

[14] J. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang, "Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, p. 8 pp., april 2005.

[15] J. Walters and V. Chaudhary, "Application-Level Checkpointing Techniques for Parallel Programs," *Distributed Computing and Internet Technology*, pp. 221–234, 2006.

[16] F. Karablieh, R. A. Bazzi, and M. Hicks, "Compiler-Assisted Heterogeneous Checkpointing," in *20th Symposium on Reliable Distributed Systems (SRDS 2001)*, pp. 56–, IEEE Computer Society, 2001.

[17] H. Jiang and V. Chaudhary, "Compile/Run-Time Support for Thread Migration," in *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, (Washington, DC, USA), pp. 100–, IEEE Computer Society, 2002.

[18] NVIDIA, *NVIDIA CUDA C Programming Guide 4.0.* 2011.

[19] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU Architecture," *IEEE Micro*, vol. 31, pp. 50–59, Mar. 2011.

[20] J. Fang, A. Varbanescu, and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," in *Parallel Processing (ICPP), 2011 International Conference on*, pp. 216 –225, sept. 2011.

[21] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2010.

[22] A. Munshi, B. Gaster, T. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming Guide*. OpenGL Series, Addison Wesley Professional, 2011.

[23] D. Schaa and D. Kaeli, "Exploring the multiple-GPU design space," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2009.

[24] D. G. Spampinato, A. C. Elster, and T. Natvig, "Modelling Multi-GPU Systems," *Science And Technology*, vol. 1, no. 6, p. 20, 2003.

[25] M. Strengert, C. Muller, C. Dachsbacher, and T. Ertl, "CUDASA: Compute Unified Device and Systems Architecture," in *EGPGV* (J. M. Favre and K.-L. Ma, eds.), pp. 49–56, Eurographics Association, 2008.

[26] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn, "Solving Dense Linear Systems on Platforms with Multiple Hardware Accelerators," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, (New York, NY, USA), pp. 121–130, ACM, 2009.

[27] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a Single Compute Device Image in OpenCL for Multiple GPUs," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, (New York, NY, USA), pp. 277–288, ACM, 2011.

[28] L. Chen, O. Villa, and G. R. Gao, "Exploring Fine-Grained Task-Based Execution on Multi-GPU Systems," in *Proceedings of the 2011 IEEE International Conference on Clus-*

ter Computing, CLUSTER '11, (Washington, DC, USA), pp. 386–394, IEEE Computer Society, 2011.

[29] B. Thies, M. Karczmarek, and S. Amarasinghe, "StreaMIT: A Language for Streaming Applications," in *In International Conference on Compiler Construction*, pp. 179–196, 2001.

[30] S. Amarasinghe, M. I. Gordon, M. Karczmarek, J. Lin, D. Maze, R. M. Rabbah, and W. Thies, "Language and Compiler Design for Streaming Applications," *Int. J. Parallel Program.*, vol. 33, pp. 261–278, June 2005.

[31] A. Hagiescu, H. P. Huynh, W.-F. Wong, and R. Goh, "Automated Architecture-Aware Mapping of Streaming Applications Onto GPUs," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pp. 467 –478, may 2011.

[32] H. P. Huynh, A. Hagiescu, W.-F. Wong, and R. S. M. Goh, "Scalable Framework for Mapping Streaming Applications onto Multi-GPU systems," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), pp. 1–10, ACM, 2012.

[33] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, (Piscataway, NJ, USA), pp. 31:1–31:11, IEEE Press, 2008.

[34] M. Lalami, D. El-Baz, and V. Boyer, "Multi GPU Implementation of the Simplex Algorithm," in *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pp. 179 –186, sept. 2011.

[35] V. Boyer, D. El Baz, and M. Elkihel, "Dense Dynamic Programming on Multi GPU," in *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pp. 545 –551, feb. 2011.

[36] L. Solano-Quinde, B. Bode, and A. K. Somani, "Techniques for the Parallelization of Unstructured Grid Applications on Multi-GPU Systems," in *Proceedings of the 2012 Interna-*

tional Workshop on Programming Models and Applications for Multicores and Manycores, PMAM '12, (New York, NY, USA), pp. 140–147, ACM, 2012.

[37] E. Attardo, A. Borsic, and R. Halter, "A Multi-GPU Acceleration for 3D Imaging of the Prostate," in *Electromagnetics in Advanced Applications (ICEAA), 2011 International Conference on*, pp. 1096 –1099, sept. 2011.

[38] B. Jang, D. Kaeli, S. Do, and H. Pien, "Multi GPU Implementation of Iterative Tomographic Reconstruction Algorithms," in *Biomedical Imaging: From Nano to Macro, 2009. ISBI '09. IEEE International Symposium on*, pp. 185 –188, 28 2009-july 1 2009.

[39] H. Kondo, E. Heien, M. Okita, D. Werthimer, and K. Hagihara, "A Multi-GPU Spectrometer System for Real-Time Wide Bandwidth Radio Signal Analysis," in *Parallel and Distributed Processing with Applications (ISPA), 2010 International Symposium on*, pp. 594 –604, sept. 2010.

[40] T. Austin, V. Bertacco, S. Mahlke, and Y. Cao, "Reliable Systems on Unreliable Fabrics," *IEEE Design Test of Computers*, vol. 25, pp. 322 –332, july-aug. 2008.

[41] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Configurable Isolation: Building High Availability Systems with Commodity Multi-Core Processors," in *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, (New York, NY, USA), pp. 470–481, ACM, 2007.

[42] W. McKee, H. McAdams, E. Smith, J. McPherson, J. Janzen, J. Ondrusek, A. Hyslop, D. Russell, R. Coy, D. Bergman, N. Nguyen, T. Aton, L. Block, and V. Huynh, "Cosmic Ray Neutron Induced Upsets as a Major Contributor to the Soft Error Rate of Current and Future Generation DRAMs," in *34th Annual Proceedings of the IEEE International Reliability Physics Symposium*, pp. 1 –6, 30 1996-may 2 1996.

[43] S. Borkar, "Design Challenges of Technology Scaling," *IEEE Micro*, vol. 19, pp. 23 –29, jul-aug 1999.

[44] Tandem and Compaq Corporation, *Data Integrity for Compaq NonStop Himalaya Servers*. White Paper, 1999.

[45] T. Slegel, I. Averill, R.M., M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, J. MacDougall, T. McPherson, J. Navarro, E. Schwarz, K. Shum, and C. Webb, "IBM's S/390 G5 microprocessor design," *Micro, IEEE*, vol. 19, pp. 12 –23, mar/apr 1999.

[46] L. Spainhower and T. A. Gregg, "IBM S/390 parallel enterprise server G5 fault tolerance: a historical perspective," *IBM Journal of Research and Development*, vol. 43, pp. 863–873, Sept. 1999.

[47] Seongwoo Kim and A. Somani, "SSD: an affordable fault tolerant architecture for superscalar processors," in *Proceedings of the Eight Pacific Rim International Symposium on Dependable Computing*, pp. 27 –34, 2001.

[48] K. Reick, P. Sanda, S. Swaney, J. Kellington, M. Mack, M. Floyd, and D. Henderson, "Fault-Tolerant Design of the IBM Power6 Microprocessor," *IEEE Micro*, vol. 28, pp. 30 –38, march-april 2008.

[49] J. Nickel and A. Somani, "REESE: A Method of Soft Error Detection in Microprocessors," in *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN)*, pp. 401 –410, July 2001.

[50] N. Azizi and P. Yiannacouras, "Gate Oxide Breakdown," Dec. 2003.

[51] A. Asenov, "Random Dopant Induced Threshold Voltage Lowering and Fluctuations in Sub 50 nm MOSFETs: A Statistical 3D 'Atomistic' Simulation Study," *Nanotechnology*, vol. 10, no. 2, p. 153, 1999.

[52] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The Impact of Technology Scaling on Lifetime Reliability," in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, (Washington, DC, USA), pp. 177–, IEEE Computer Society, 2004.

[53] F. A. Bower, D. J. Sorin, and S. Ozev, "A Mechanism for Online Diagnosis of Hard Faults in Microprocessors," in *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, (Washington, DC, USA), pp. 197–208, IEEE Computer Society, 2005.

[54] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed Design and Evaluation of Redundant Multithreading Alternatives," *SIGARCH Computer Architecture News*, vol. 30, pp. 99–110, May 2002.

[55] T. M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," in *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 32, (Washington, DC, USA), pp. 196–207, IEEE Computer Society, 1999.

[56] *Top 500 Supercomputer Sites*, Nov. 2012. Available: `http://www.top500.org/lists/2012/11`.

[57] ORNL.GOV, *Introducing Titan: Advancing the Era of Accelerated Computing.* Oak Ridge National Laboratory. `http://www.olcf.ornl.gov/titan/`.

[58] AMD, *AMD Optetron Processors.* Oak Ridge National Laboratory. Available: `http://www.nccs.gov/wp-content/uploads/2010/02/AMD-5.09.10.pdf`.

[59] X.-J. Yang, X.-K. Liao, K. Lu, Q.-F. Hu, J.-Q. Song, and J.-S. Su, "The TianHe-1A Supercomputer: Its Hardware and Software," *Journal of Computer Science and Technology*, vol. 26, pp. 344–351, 2011. 10.1007/s02011-011-1137-8.

[60] P. Gepner, M. F. Kowalik, D. L. Fraser, and K. Wackowski, "Early Performance Evaluation of New Six-Core Intel Xeon 5600 Family Processors for HPC," in *Proceedings of the Ninth International Symposium on Parallel and Distributed Computing (ISPDC)*, pp. 117 –124, july 2010.

[61] L. Solano-Quinde and B. Bode, "RAS and Job Log Data Analysis for Failure Prediction for the IBM Blue Gene/L," in *Proceedings of the Parallel and Distributed Computing and Systems (PDCS 2008)*, Nov 2008.

[62] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam, "Critical Event Prediction for Proactive Management in Large-Scale Computer Clusters," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, (New York, NY, USA), pp. 426–435, ACM, Aug. 2003.

[63] R. K. Sahoo, M. Bae, R. Vilalta, J. Moreira, S. Ma, and M. Gupta, "Providing Persistent and Consistent Resources through Event Log Analysis and Predictions for Large-scale Computing Systems," in *Workshop on Self-Healing, Adaptive and SelfMANaged Systems (SHAMAN)*, June 2002.

[64] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under UNIX," in *USENIX Winter'95*, pp. 213–224, Jan. 1995.

[65] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System," Technical Report CS-TR-1997-1346, University of Wisconsin, Madison, Apr. 1997.

[66] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz, "Application-Level Checkpointing for Shared Memory Programs," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, (New York, NY, USA), pp. 235–247, ACM, 2004.

[67] H. Zhong and J. Nieh, "CRAK: Linux Checkpoint/Restart as a Kernel Module," Technical Report CUCS-014-01, Department of Computer Science - Columbia University, Nov. 2001.

[68] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis, "Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, (Washington, DC, USA), pp. 9–, IEEE Computer Society, 2005.

[69] A. J. Oliner, R. K. Sahoo, J. E. Moreira, and M. Gupta, "Performance Implications of Periodic Checkpointing on Large-Scale Cluster Systems," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18 - Volume 19*, IPDPS '05, (Washington, DC, USA), pp. 299.2–, IEEE Computer Society, 2005.

[70] C. M. Krishna, Y.-H. Lee, and K. G. Shin, "Optimization Criteria for Checkpoint Placement," *Communications of the ACM*, vol. 27, pp. 1008–1012, Oct. 1984.

[71] Y. Ling, J. Mi, and X. Lin, "A Variational Calculus Approach to Optimal Checkpoint Placement," *IEEE Transaction on Computers*, vol. 50, pp. 699–708, July 2001.

[72] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "CheCUDA: A Checkpoint/Restart Tool for CUDA Applications," in *10-th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pp. 408 –413, dec. 2009.

[73] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters," *Journal of Physics: Conference Series*, vol. 1, no. 46, pp. 494–499, 2006.

[74] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi, "CheCL: Transparent Checkpointing and Process Migration of OpenCL Applications," in *2011 IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pp. 864 –876, may 2011.

[75] S. Laosooksathit, N. Naksinehaboon, C. Leangsuksan, A. Dhungana, C. Chandler, K. Chanchio, and A. Farbin, "Lightweight Checkpoint Mechanism and Modeling in GPGPU Environment," in *4th Workshop on System-level Virtualization for High Performance Computing (HPCVirt 2010)*, April 2010.

[76] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the Parallel Computing Landscape," *Commun. ACM*, vol. 52, pp. 56–67, Oct. 2009.

[77] Z. J. Wang and H. Gao, "A Unifying Lifting Collocation Penalty Formulation Including the Discontinuous Galerkin, Spectral Volume/Difference Methods for Conservation Laws on Mixed Grids," *J. Comput. Phys.*, vol. 228, pp. 8161–8186, November 2009.

[78] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers*. Cambridge, MA, USA: MIT Press, 1989.

[79] D. Culler, J. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st ed., 1998. The Morgan Kaufmann Series in Computer Architecture and Design.

[80] PCI SIG, "PCI Express Base 2.x Specification." `http://www.pcisig.com/specifications/pciexpress/base2/`, 2011.

[81] A. K. Somani and A. M. Sansano, "Achieving Robustness and Minimizing Overhead in Parallel Algorithms Through Overlapped Communication/Computation," *J. Supercomput.*, vol. 16, pp. 27–52, May 2000.

[82] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering*, vol. 12, pp. 66–73, May 2010.

[83] J. Owens, "Streaming Architectures and Technology Trends," in *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, (New York, NY, USA), ACM, 2005.

[84] K.-H. Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. Comput.*, vol. 33, pp. 518–528, June 1984.

[85] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '04, (New York, NY, USA), pp. 133–137, ACM, 2004.

[86] T. Ayav, P. Fradet, and A. Girault, "Implementing Fault-Tolerance in Real-Time Systems by Automatic Program Transformations," in *Proceedings of the 6th ACM & IEEE*

International conference on Embedded software, EMSOFT '06, (New York, NY, USA), pp. 205–214, ACM, 2006.

[87] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated Application-Level Checkpointing of MPI Programs," in *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, (New York, NY, USA), pp. 84–94, ACM, 2003.

[88] L. Solano-Quinde, B. Bode, and A. Somani, "Coarse Grain Computation-Communication Overlap for Efficient Application-Level Checkpointing for GPUs," in *Electro/Information Technology (EIT), 2010 IEEE International Conference on*, pp. 1 –5, may 2010.

[89] J. W. Young, "A First Order Approximation to the Optimum Checkpoint Interval," *Commun. ACM*, vol. 17, pp. 530–531, Sept. 1974.

[90] J. Daly, "A Model for Predicting the Optimum Checkpoint Interval for Restart Dumps," in *Proceedings of the 2003 international conference on Computational science*, ICCS'03, (Berlin, Heidelberg), pp. 3–12, Springer-Verlag, 2003.

[91] A. J. Oliner, L. Rudolph, and R. K. Sahoo, "Cooperative Checkpointing: A Robust Approach to Large-Scale Systems Reliability," in *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, (New York, NY, USA), pp. 14–23, ACM, 2006.

[92] L. Solano-Quinde, Z. J. Wang, B. Bode, and A. K. Somani, "Unstructured grid applications on GPU: performance analysis and improvement," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, (New York, NY, USA), pp. 13:1–13:8, ACM, 2011.