


Winter 2014

Performance modeling and optimization techniques for heterogeneous computing

Supada Laosooksathit

Follow this and additional works at: <https://digitalcommons.latech.edu/dissertations>

 Part of the [Applied Statistics Commons](#), [Mathematics Commons](#), and the [Other Computer Sciences Commons](#)

**PERFORMANCE MODELING AND OPTIMIZATION TECHNIQUES
FOR HETEROGENEOUS COMPUTING**

by

Supada Laosooksathit, B.S., M.S.

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

COLLEGE OF ENGINEERING AND SCIENCE
LOUISIANA TECH UNIVERSITY

March 2014

UMI Number: 3662202

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI 3662202

Published by ProQuest LLC 2015. Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code.



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

LOUISIANA TECH UNIVERSITY

THE GRADUATE SCHOOL

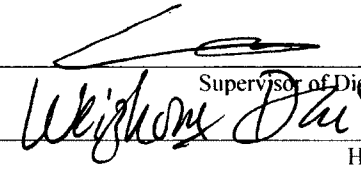
Dec 10, 2013

Date

We hereby recommend that the dissertation prepared under our supervision
by SUPADA LAOSOOKSATHIT

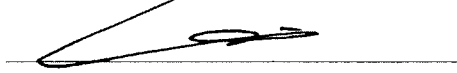
entitled PERFORMANCE MODELING AND OPTIMIZATION TECHNIQUES
FOR HETEROGENEOUS COMPUTING

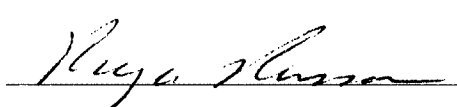
be accepted in partial fulfillment of the requirements for the Degree of
DOCTOR OF PHILOSOPHY





Supervisor of Dissertation Research
Head of Department
Computational Analysis and Modeling
Department

Recommendation concurred in:






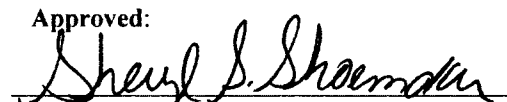




Advisory Committee

Approved:


Director of Graduate Studies

Approved:


Dean of the Graduate School



Dean of the College

ABSTRACT

Since Graphics Processing Units (GPUs) have increasingly gained popularity among non-graphic and computational applications, known as General-Purpose computation on GPU (GPGPU), GPUs have been deployed in many clusters, including the world's fastest supercomputer. However, to make the most efficiency from a GPU system, one should consider both performance and reliability of the system.

This dissertation makes four major contributions. First, the two-level checkpoint/restart protocol that aims to reduce the checkpoint and recovery costs with a latency hiding strategy in a system between a CPU (Central Processing Unit) and a GPU is proposed. The experimental results and analysis reveals some benefits, especially in a long-running application.

Second, a performance model for estimating GPGPU execution time is proposed. This performance model improves operation cost estimation over existing ones by considering varied memory latencies. The proposed model also considers the effects of thread synchronization functions. In addition, the impacts of various issues in GPGPU programming such as bank conflicts in shared memory and branch divergence are also discussed.

Third, the interplay between GPGPU application performance and system reliability of a large GPU system is explored. This includes a checkpoint scheduling

model for a certain GPGPU application. The effects of a checkpoint/restart mechanism on the application performance is also discussed.

Finally, optimization techniques to remedy uncoalesced memory access in GPU's global memory are proposed. These techniques are memory rearrangement using 2-dimensional matrix transpose and 3-dimensional matrix permutation. The analytical results show that the proposed technique can reduce memory access time, especially when the transformed array/matrix is frequently accessed.

APPROVAL FOR SCHOLARLY DISSEMINATION

The author grants to the Prescott Memorial Library of Louisiana Tech University the right to reproduce, by appropriate methods, upon request, any or all portions of this Dissertation. It is understood that "proper request" consists of the agreement, on the part of the requesting party, that said reproduction is for his personal use and that subsequent reproduction will not occur without written approval of the author of this Dissertation. Further, any portions of the Dissertation used in books, papers, and other works must be appropriately referenced to this Dissertation.

Finally, the author of this Dissertation reserves the right to publish freely, in the literature, at any time, any or all portions of this Dissertation.

Author Supada Laosooksathit



Date 12/10/2013

DEDICATION

To my late mother, Suvimol Laosooksathit, I dedicate this work.

TABLE OF CONTENTS

| | |
|--|-----|
| ABSTRACT | iii |
| DEDICATION | vi |
| LIST OF TABLES..... | x |
| LIST OF FIGURES..... | xi |
| ACKNOWLEDGMENTS | xv |
| CHAPTER 1 INTRODUCTION | 1 |
| 1.1 Overview of GPUs | 1 |
| 1.2 GPGPU Performance and Optimization..... | 2 |
| 1.3 Checkpoint/Restart Mechanism..... | 4 |
| CHAPTER 2 BACKGROUND AND RELATED WORKS | 6 |
| 2.1 Checkpoint/Restart Mechanism..... | 6 |
| 2.2 Checkpoint Scheduling and System Reliability | 7 |
| 2.3 GPGPU Performance Models | 8 |
| 2.4 Memory Optimization..... | 9 |
| 2.4.1 Memory Optimization in HPC | 9 |
| 2.4.2 Memory Optimization in GPGPU..... | 11 |
| 2.4.2.1 GPU-CPU Memory Latency Hiding..... | 11 |
| 2.4.2.2 GPGPU Memory Hierarchy..... | 12 |
| CHAPTER 3 TWO-LEVEL CHECKPOINT/RESTART FOR GPGPU..... | 14 |

| | | |
|--|--|----|
| 3.1 | CUDA Streams | 14 |
| 3.1.1 | Performance Model of CUDA Streams..... | 16 |
| 3.1.2 | Benchmarks | 22 |
| 3.2 | Two-Level Checkpoint/Restart Protocols | 25 |
| 3.3 | Experiments | 27 |
| 3.4 | Simulation | 31 |
| 3.5 | Conclusion..... | 35 |
| CHAPTER 4 PERFORMANCE MODEL FOR GPGPU | | 36 |
| 4.1 | Performance Modelling | 38 |
| 4.1.1 | Parameters..... | 38 |
| 4.1.2 | Performance Model in a General Case | 42 |
| 4.1.3 | Performance Model in a Case with Synchronization | 43 |
| 4.1.4 | Control Flow Cases..... | 45 |
| 4.2 | Results and Discussion..... | 47 |
| 4.2.1 | Code Analysis..... | 47 |
| 4.2.2 | Results..... | 49 |
| CHAPTER 5 GPU APPLICATION PERFORMANCE VS CHECKPOINTS ... | | 52 |
| 5.1 | Application Performance and System Reliability..... | 53 |
| 5.1.1 | Predicting the System Size from the Maximum System Reliability | 57 |
| 5.1.2 | Predicting the System Size from an Expected Performance..... | 58 |
| 5.1.2.1 | Non-scalable Workload | 59 |
| 5.1.2.2 | Scalable Workload | 59 |
| 5.2 | Checkpoint Scheduling..... | 60 |

| | | |
|---|---|----|
| 5.3 | Performance | 64 |
| 5.4 | Real-World Case Study | 69 |
| 5.5 | The Model with an Excess Weibull | 72 |
| 5.6 | Conclusion | 74 |
| CHAPTER 6 GPGPU OPTIMIZATION | | 75 |
| 6.1 | Coalescing VS Uncoalescing | 76 |
| 6.2 | Memory Rearrangement | 78 |
| 6.2.1 | Single Stride | 79 |
| 6.2.2 | Double Strides | 81 |
| 6.3 | Analytical Models | 86 |
| 6.3.1 | Predetermined Data Size | 86 |
| 6.3.2 | Unknown Data Size | 88 |
| 6.3.2.1 | Single Stride | 88 |
| 6.3.2.2 | Double Strides | 90 |
| 6.4 | Results and Cost Analysis | 90 |
| 6.4.1 | Predetermined Data Size | 91 |
| 6.4.2 | Unknown Data Size and Break-even Analysis | 91 |
| 6.5 | Conclusion | 92 |
| CHAPTER 7 CONCLUSIONS AND FUTURE WORK | | 94 |
| BIBLIOGRAPHY | | 96 |

LIST OF TABLES

| | | |
|------------|--|----|
| Table 3.1: | Times spent by operations of matrix multiplication, measured by <i>CUDA Events</i> | 23 |
| Table 3.2: | Total execution times with non-stream and the expected total execution time with 8-stream are calculated from the operational times on Table 3.1..... | 24 |
| Table 4.1: | The values of experimental parameters validated in the model..... | 48 |
| Table 6.1: | The comparison between the kernel execution times based on our technique using the sample code in Listing 6.4 with the original memory access and transformed memory access..... | 91 |
| Table 6.2: | The comparison between the performance gain from the transformed memory access to the time of 2D matrix transpose..... | 92 |

LIST OF FIGURES

| | |
|--|----|
| Figure 1.1: Two-level checkpoint for a heterogeneous system | 5 |
| Figure 2.1: Memory hierarchy in GPGPU | 12 |
| Figure 3.1: The results of simpleStreams application with various numbers of streams and array sizes | 15 |
| Figure 3.2: Time diagram of non-streamed execution | 17 |
| Figure 3.3: Time diagram of streamed execution: (a) asynchronous memory copies from host to device, (b) asynchronous memory copies from device to host, and (c) asynchronous memory copies in both directions. | 18 |
| Figure 3.4: Time diagram of execution time when the time of memory copies are longer than kernel executions: (a) synchronous memory copies, (b) synchronous memory copies from host to device, (c) asynchronous memory copies from device to host, and (d) asynchronous memory copies in both directions. | 20 |
| Figure 3.5: Time diagram of execution time when the time of kernel executions are longer than memory copies: (a) synchronous memory copies, (b) synchronous memory copies from host to device, (c) asynchronous memory copies from device to host, and (d) asynchronous memory copies in both directions. | 21 |
| Figure 3.6: The checkpoint protocol for GPU streamed CPR..... | 26 |
| Figure 3.7: The restart protocol for GPU streamed CPR..... | 27 |
| Figure 3.8: The percentage of performance improvement in terms of (a) checkpoint cost, (b) recovery cost due to a failure occurrence, and (c) wasted time due to a failure occurrence..... | 30 |
| Figure 3.9: The percentages of total checkpoint costs compared to wasted time of the application with non-streamed, 4-streamed, and 8-streamed CPRs, when the size of arrays is 2^{24} and (a) the checkpoint interval is 30 minutes, and (b) the MTTF is 12 hours. | 32 |

| | |
|---|----|
| Figure 3.10: The percentages of total recovery costs compared to wasted time of the application with non-streamed, 4-streamed, and 8-streamed CPRs, when the size of arrays is 2^{24} and (a) the checkpoint interval is 30 minutes, and (b) the MTTF is 12 hours. | 33 |
| Figure 3.11: The percentages of wasted time compared to completion time of the application with non-streamed, 4-streamed, and 8-streamed CPRs, when the size of arrays is 2^{24} and (a) the checkpoint interval is 30 minutes, and (b) the MTTF is 12 hours. | 33 |
| Figure 4.1: Memory latency on GPU NVIDIA GeForce GTX 295 | 37 |
| Figure 4.2: GPU architecture: (a) programming model; (b) warp scheduling in an SM | 39 |
| Figure 4.3: The instruction list in (a) transforms into the timeline in (b). | 41 |
| Figure 4.4: Diagrams show the execution time when: (a) there is no idle time; (b) there is idle time. A shaded box indicates that the warp is being executed. A white box indicates that the warp is waiting for the memory access. | 43 |
| Figure 4.5: Diagrams show the execution time when: (a) a synchronization function causes extra time between two consecutive computational instructions; (b) a synchronization function causes extra time between memory and computational instructions. Again, a shaded box indicates that the warp is being executed. A white box indicates that the warp is waiting for the memory access. | 44 |
| Figure 4.6: The kernel execution time from the performance model compared to the measurement of: (a) naïve matrix multiplication; (b) tiled matrix multiplication | 49 |
| Figure 4.7: Percentage of error of the performance model compared to the measurement for both benchmarks | 50 |
| Figure 5.1: Probability of survival for different values of c and k from Equation (5.4) | 56 |
| Figure 5.2: The time between two consecutive checkpoints when: (a)–(c) k is predicted by the maximal reliability expressed by Equation (5.8); (d)–(f) k is predicted by the desired performance with fixed workload expressed by Equation (5.9); (g)–(i) k is predicted by the desired performance with scalable workload expressed by Equation (5.10). ... | 64 |

| | |
|--|----|
| Figure 5.3: Comparison of application performance when: (a)–(c) k is predicted by the maximal reliability expressed by Equation (5.8); (d)–(f) k is predicted by the desired performance with fixed workload expressed by Equation (5.9); (g)–(i) k is predicted by the desired performance with scalable workload expressed by Equation (5.10)..... | 68 |
| Figure 5.4: Time between two consecutive checkpoints for the case study when: (a)–(c) k is predicted by the maximal reliability expressed by Equation (5.8); (d)–(f) k is predicted by the desired performance with fixed workload expressed by Equation (5.9); (g)–(i) k is predicted by the desired performance with scalable workload expressed by Equation (5.10)..... | 70 |
| Figure 5.5: Comparison of application performance for the case study when: (a)–(c) k is predicted by the maximal reliability expressed by Equation (5.8); (d)–(f) k is predicted by the desired performance with fixed workload expressed by Equation (5.9); (g)–(i) k is predicted by the desired performance with scalable workload expressed by Equation (5.10)..... | 71 |
| Figure 6.1: Memory access pattern categories: (a) Coalesced memory access, (b) Single stride uncoalesced memory access, and (c) Double stride uncoalesced memory access..... | 77 |
| Figure 6.2: Array B transformation associated to the sample code in Listing 6.4 and 6.5 (a) The original array; (b) The 2D matrix is constructed; (c) The transformed array..... | 80 |
| Figure 6.3: Transformation of the 2D matrix that only the elements on the diagonal line are accessed as given by Listing 6.7; (a) Original Layout, (b) Modified matrix with the width of $n_B + 1$ | 81 |
| Figure 6.4: 3D Matrix | 82 |
| Figure 6.5: 1D array with double strides where $d_i < d_j$; (a) The original layout, (b) The 3D matrix with $m = d_i$, $n = d_j/d_i$, (c) Permuted 3D matrix with the order of [2,3,1], and (d) The final layout..... | 83 |
| Figure 6.6: 1D array with double strides where $d_i > d_j$; (a) The original layout, (b) The 3D matrix with $m = d_i/d_j$, $n = d_j$, (c) Permuted 3D matrix with the order of [2,1,3], and (d) The final layout..... | 84 |
| Figure 6.7: 2D matrix with double strides; (a) The original layout, (b) The final layout after 3D matrix permutation with the order of [2,1,3]..... | 85 |

Figure 6.8: Tiled 2D matrix transposition where (a) is the original matrix and
(b) is the transposed matrix..... 88

ACKNOWLEDGMENTS

This dissertation would not have been possible without the great advice of my advisor, Dr. Chokchai (Box) Leangsuksun, who has been patiently supporting me throughout my doctoral study. He has guided me in every aspect of my research and career, including writing and presenting research, providing critical feedback, giving me opportunities to work with several researchers from many scientific areas, and encouraging me to always move forward.

I would like to acknowledge Dr. Raja Nassar for his guidance in probability and statistics and his patience in editing the technical papers. I am sincerely grateful to Dr. Mihaela Paun for discussing ideas and validating my research in statistics, Dr. Weizhong Dai for advising me through the CAM program, and Dr. Zeno Greenwood for his valuable comments.

In personal, I would like to thank Nichamon Naksinehaboon and Narate Taerat for helping me throughout the study and for living in Ruston. I would also like to thank Thanadech Thanakornworakij for giving moral suggestions during my hard times. Most importantly, I would like to express my deepest gratitude to my mother, Suvimol Laosooksathit, who always taught me to be patient and persistent; my father, Dr. Surin Laosooksathit, who is my role model to persue a doctoral degree and always loves and supports me unconditionally; and my sister, Wipada Laosooksathit, who is my morale and always believes in me.

CHAPTER 1

INTRODUCTION

Exascale computing demands higher computational power and massive parallelism. In response to this demand, many organizations have upgraded their computational infrastructures. Oak Ridge National Laboratory (ORNL)'s Titan has also been upgraded and has become the world's fastest supercomputer. Titan's computational power is mainly from the newest NVIDIA's graphic cards [1]. Therefore, to make the most efficiency from a system with Graphics Processing Units (GPUs), the application performance on such a system and the GPU system reliability should be carefully studied.

1.1 Overview of GPUs

GPUs were first introduced for graphics computation. Due to the ability to accelerate computation by massive data-parallelism, GPUs have been deployed for non-graphic applications, known as General-Purpose computation on GPU (GPGPU) [2]. The paradigm to exploit both CPU (Central Processing Unit) and GPU is also known as heterogeneous computing.

For a node-wise system, since a GPU works as a co-processor of a CPU, the CPU is referred to as the host, and the GPU as the device. First the data set has to be prepared on the host side and transferred to the device. This process is called

host-to-device memory copy. Once host-to-device memory copy finishes, the program is executed on the device side. A part of the program that will be executed on the device is called a kernel. After the kernel execution finishes, the results are transferred back to the host. This process is called device-to-host memory copy [3] [4]. Note that a host thread can handle only one GPU device.

Once the kernel is invoked, a kernel grid is created inside the GPU. The grid contains thread blocks that are distributed to the GPU streaming multiprocessors (SMs). Each thread block contains a number of threads that will execute instructions on an SM. A new set of thread blocks is launched on the SMs again as previous thread blocks terminate [3].

1.2 GPGPU Performance and Optimization

By estimating cost benefit of GPGPU computing, the programmers will be able to find ways to optimize their parallel program for a better use of both CPU and GPU in a single application. Therefore, a performance model for GPGPU has become more crucial in order to gain an insight into speed improvement issues from a High Performance Computing (HPC) software development perspective.

In order to estimate an application completion-time, the memory transfer time between the GPU and the CPU, and the kernel execution time must be obtained. Thus, one can make a decision whether the GPU is worth participating in the computation and can further improve the heterogeneous computing application.

Furthermore, the performance model can also help to improve the efficiency of fault tolerance techniques for GPGPU, such as checkpoint/restart mechanisms,

especially for a large GPU cluster. One of the major problems in this area is checkpoint scheduling. To find an optimum checkpoint placement, the time-to-failure (TTF) of the system and the completion-time of the application must be taken into account. The system TTF can be obtained by a failure prediction technique, while the completion time of the application can be estimated by the performance model. The details of the performance model for a GPGPU application is illustrated in Chapter 4.

Even though deploying many computing elements can increase the computing power, a very large system is prone to fail. Both soft and hard failures can cause interruptions to applications running on the system at that time. Thus, Chapter 5 will describe an interplay between the performance of a GPGPU application and the reliability of a GPU system. The model to find an optimal number of nodes that will satisfy both criteria is also proposed.

In addition, an optimization technique that aims toward performance improvement in GPGPU applications is presented in Chapter 6. Since one of the major concerns in GPGPU optimization is coalescing in global memory, this work proposes memory rearrangement techniques to remedy uncoalesced global memory access patterns by using 2-dimensional matrix transpose and 3-dimensional matrix permutation. The proposed techniques can be applied to many common memory access patterns. The cost benefit of these techniques will also be discussed in details. Moreover, the analytical results reveal that the proposed techniques are beneficial if the transformed array/matrix is frequently accessed.

1.3 Checkpoint/Restart Mechanism

Since a large-scale GPU cluster is the main focus system in this dissertation, fault tolerance is an important issue that should not be omitted.

Checkpoint/restart is a fault tolerance mechanism that has been used in many system platforms. Instead of restarting computation from the beginning when a failure occurs, with checkpoint/restart, a process can be rolled back from the last checkpoint and can be migrated to a healthier node or system [5] [6].

However, GPU fault tolerance has recently been a concern in HPC. There are very few existing works that address resilience issues in a GPU environment. However, with the popularity of large GPU systems, it is anticipated that reliability will be quite critical for its future success.

In a GPU system, any failure that occurs during kernel execution will normally cause a loss of kernel computation. Figure 1.1 shows a two-step checkpoint protocol that first transfers checkpoint data and GPU status from the GPU to the CPU memory and then saves the software state to either a reliable storage or a healthier node.

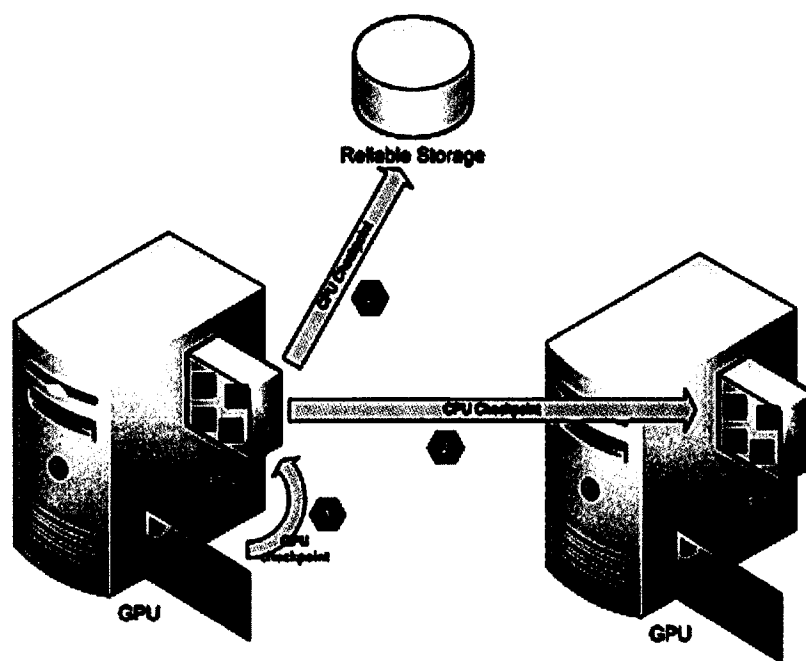


Figure 1.1: Two-level checkpoint for a heterogeneous system

To improve the utilization of the checkpoint/restart mechanism, there are two major concerns: reducing the costs of checkpoint and recovery processes, and finding optimal checkpoint placements. To reduce the costs of checkpoint and recovery processes, the two-level checkpoint/restart protocols are further discussed in Chapter 3. These protocols utilize a latency hiding strategy to reduce the memory transfer time. Moreover, the checkpoint scheduling model for finding a sequence of optimal checkpoint placements is presented in Chapter 5. The proposed model aims to reduce the wasted time, which is an aggregation of checkpoint costs, recovery costs, and recomputing time, while increasing the application performance in case of failure.

CHAPTER 2

BACKGROUND AND RELATED WORKS

This chapter discusses the related work on a checkpoint/restart mechanism, performance model, and application optimization for GPGPU.

2.1 Checkpoint/Restart Mechanism

A checkpoint/restart mechanism is a process to improve the application resilience. By saving the application state and considerable data in the checkpoint file, the process can be restarted from that state later. Therefore, the recomputing time – the time spent to re-execute the work due to a failure – can be reduced. There are many existing works that have been implemented for checkpoint/restart [5] [7] [8] [9].

BLCR [8] is a checkpoint/restart mechanism for Linux systems. It is developed for checkpointing at the operating system-level, which allows the system preemption. Periodic and preemptive checkpointing can be used in response to the precursors of a possible failure.

VCCP [5] provides a transparent checkpoint/restart mechanism for virtual machines (VMs). It uses a hypervisor-based coordinated checkpoint/restart protocol so that the guest OS does not have to be changed.

For GPGPU resiliency, CheCUDA [6] is a checkpoint/restart mechanism for NVIDIA's CUDA (Compute Unified Device Architecture). However, it results in

reduction of system performance due to the checkpoint overhead, particularly, for a large data set. This cost is mainly produced by memory transfer.

HiAL-Ckpt [10] is also a checkpoint/restart mechanism for GPGPU. However, it is implemented based on Brook+ programming language and allows the programmer to do checkpoints at the application level. Although its idea of the hierarchical checkpoint is similar to our previous work [11], the overhead optimization is not considered.

2.2 Checkpoint Scheduling and System Reliability

There have been existing checkpoint scheduling models that aim to minimize wasted time. Liu *et al* [12] have proposed a scheme to derive a sequence of checkpoint placements that uses the theory of stochastic renewal reward process. Although their model targets a large-scale HPC system, it can be applied to any system as long as the system reliability is derivable.

Paun *et al* [13] have presented a scheduling model for incremental checkpoints. An incremental checkpoint mechanism has been introduced to reduce the checkpoint cost. They have proposed a scheme to find an optimal number of incremental checkpoints for any failure intensity function.

Since the aforementioned checkpoint scheduling models rely on a system reliability model, for an HPC system, Gottumukkala *et al* [14] have proposed a reliability model of a system of k nodes where each individual node follows a Weibull distribution. Their model also considers the excess life of survival nodes. The results have shown that their model is more accurate than previous models used in literature.

Thanakornworakij *et al* [15] have proposed a new reliability model that is an extension of Gottumukkala's work [14]. Their work is based on the reliability of a system of k nodes with simultaneous failures. Their model considers the correlation between nodes, which can make the nodes in the system fail simultaneously.

2.3 GPGPU Performance Models

Today, as GPUs have become popular in the HPC area, there have been existing works that model GPGPU performance by estimating the kernel execution time [16] [17] [18] [19].

Hong and Kim [16] have introduced two metrics, Memory Warp Parallelism (MWP) and Computation Warp Parallelism (CWP) in order to describe the GPU parallel architecture. This performance model aims to predict the GPGPU performance from CPU code skeletons.

Zhang and Owens [18] have developed a quantitative performance model based on their microbenchmarks so that they can identify bottlenecks in the program. Nevertheless, this model does not incorporate the cache model, bank-conflict, thread synchronization, and non-perfect pipeline in an SM.

Baghsorkhi *et al* [19] have presented an analytical model to predict the performance based on a GPGPU work flow graph. The model allows a compiler to determine the benefit of parallelization mathematically. Moreover, the model can identify the bottlenecks to guide the compiler through the optimization process.

Furthermore, Wong *et al* [20] have developed a suite of microbenchmarks to obtain the architectural characteristics of an NVIDIA's GPU. They have also investigated the architectural details of the processing cores and the memory hierarchies.

2.4 Memory Optimization

In scientific and mathematical problem domains, a lot of application data sets are organized and operated as arrays and matrices. For example, iterative methods are commonly used for solving scientific problems. However, those methods require high data communication, which may cause bottlenecks in parallelization [21] [22].

2.4.1 Memory Optimization in HPC

There have been many attempts to optimize the cost of data communication. Data alignment problems are an issue that aims to assign data and computations to a set of virtual processors [23] [24]. David Bau *et al* [23] have proposed a strategy to solve alignment problem by using elementary linear algebra. They have claimed that their strategy is able to achieve communication-free alignment.

In a distributed environment, such as grid computing, the data are scattered in different locations. Chervenak *et al* [25] have discussed the effect of data placement policies and workflow management systems in a grid environment. Ranganathan *et al* [26] have presented a data scheduling framework and data movement operations that address resource utilization, response time, resource locality, etc.

However, those strategies may not be able to directly apply to a co-processor environment such as GPGPU. In GPGPU, the data are resided in a memory unit

called global memory. There have been many studies that aim to reduce the global memory access time.

Yang *et al* [27] have introduced an optimization compiler for GPGPU. Their compiler analyzes off-chip memory access patterns and optimizes the memory accesses through vectorization and coalescing by using shared memory. Then it analyzes data dependencies and identifies possible data sharing across threads and thread blocks and merges threads and/or thread blocks to improve memory reuse. After that, it uses a data prefetching technique from temporary variables to overlap global memory latency with computation.

Bader *et al* [28] have proposed a CUDA library for a set of data rearrangement operations. They address four types of generic kernels. (1) The first type consists of basic read/write routines. They are kernels that provide optimal read/write accesses from the global memory by allowing for data transfer as per common access patterns. (2) The second type consists of data reordering routines, which are kernels for rearranging N -dimensional array into M -dimensional array, where $M \leq N$, by employing an offset/striding approach and shared memory. (3) The third are interlace and de-Interlace kernels. An interlace kernel combines multiple data-sets to form a single data-set. A de-interlace kernel splits a single data-set into multiple smaller data-sets. (4) Forth, a generic stencil computation kernel provides a generic and optimal framework for stencil computations, where each point in a 2D grid is updated with weighted contributions from its neighbors (e.g. in PDE solvers).

The techniques proposed in Chapter 6 aim to achieve an optimization at the compile-time under a condition that the data size is predetermined. In the case that

the data size is unknown before the run-time, matrix transpose/permutation during the compile-time is impossible. Hence, the proposed techniques must be applied during run-time with a condition of the trade-off between optimization overhead versus performance gain, i.e. the overall uncoalesced memory access time to such a data set is longer than the time for matrix transformation.

2.4.2 Memory Optimization in GPGPU

Since GPGPU performance has become crucial, especially for big-data computing, the techniques to minimize memory latency due to memory access at any level are essential in GPGPU computing.

2.4.2.1 GPU-CPU Memory Latency Hiding

Masuhara *et al* [29] have described that the latency hiding is a technique to eliminate time to wait for the remote message by overlapping local computation and remote communication. They show two versions of the sample function, which are different in the number of requests that are sent in advance of the actual use of the data. One only requests for the element that is used in the next iteration. Another one requests for all the elements before the computation. This paper also shows how the mechanisms are implemented.

NVIDIA's CUDA has introduced a latency hiding technique, called stream. Since the GPU is idle during the memory transfer, this technique allows overlapping between kernel execution and memory transfer [3]. The details of CUDA stream are discussed in Section 3.1.

2.4.2.2 GPGPU Memory Hierarchy

When a part of the program that runs on the GPU is invoked, a collection of threads called “grid” is created. There are a specific number of thread blocks in a grid, and each block has a specific number of threads. The threads in a block are managed in a group of threads call a warp. Threads in a warp will dispatch an instruction and access the memory simultaneously. Furthermore, there are multiple memory spaces in GPGPU as illustrated in Figure 2.1.

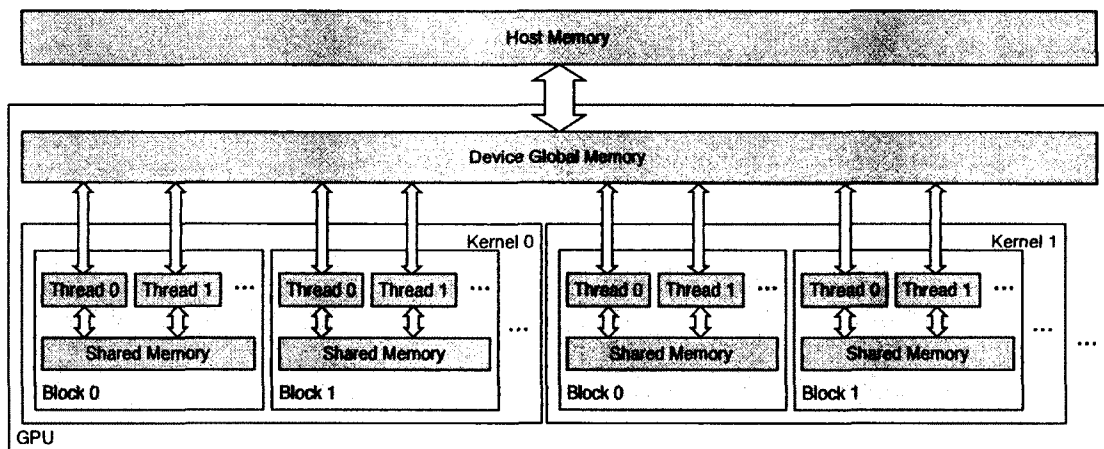


Figure 2.1: Memory hierarchy in GPGPU

In Figure 2.1, global memory is accessible by all threads in the kernel, and even across kernels. The data to be executed by the kernel have to be transferred from the CPU (host) memory and stored in the GPU (device) global memory. Once the threads in a warp are active, the data required by those threads will be simultaneously read from the global memory to the register memory. Likewise, the results from the active threads will be written to the global memory at the same time. Besides the

registers, shared memory is visible to all threads in the same block as long as the block is active. This memory unit acts as a cache where a bulk of data read from the global memory can be temporarily stored.

CHAPTER 3

TWO-LEVEL CHECKPOINT/RESTART FOR GPGPU

In a very large scale system, a fault tolerance is critical to mitigate the failures in a long running application. Checkpoint/restart is of popular fault tolerance techniques. By saving data and essential information e.g., software state, into a checkpoint file, the system can be recovered from the saved software state instead of recomputing from the beginning of the program.

This chapter presents the checkpoint/restart mechanism for GPGPU. The idea centers around transferring checkpoint data first from the GPU (device) memory to the CPU (host) memory and then from the CPU either to a reliable storage or another healthier node when a failure occurs. The saved software state will be used for recovery. The proposed checkpoint/restart protocol on the GPU utilizes the latency hiding strategy in order to improve overall performance.

3.1 CUDA Streams

NVIDIA's CUDA has introduced a latency hiding technique called CUDA stream that enables an overlap between the memory copy and kernel execution [3]. The performance models in [4] show that the stream technique can reduce the kernel execution times while the applications may require more data transfer between device and host. Moreover, it suggests that the stream technique is more suitable for the

applications in which the data are independent so that both host-to-device and device-to-host memory transfer can be carried on without kernel interruption.

In NVIDIA's CUDA, streams usage is illustrated by an application called simpleStreams. The application initializes an array, each assigned a specific value.

In the experiment that renders the results as shown in Figure 3.1, each value in the array is initialized as an integer, for example 5. The host and the device memories are then allocated to the array of size n , which varies from 64 thousand to 16 million integers. Then the integer and the array are synchronously copied to the device memory. In addition, the number of streams (s) varies between 1, 2, 4, 8, 16, 32, 64, and 128.

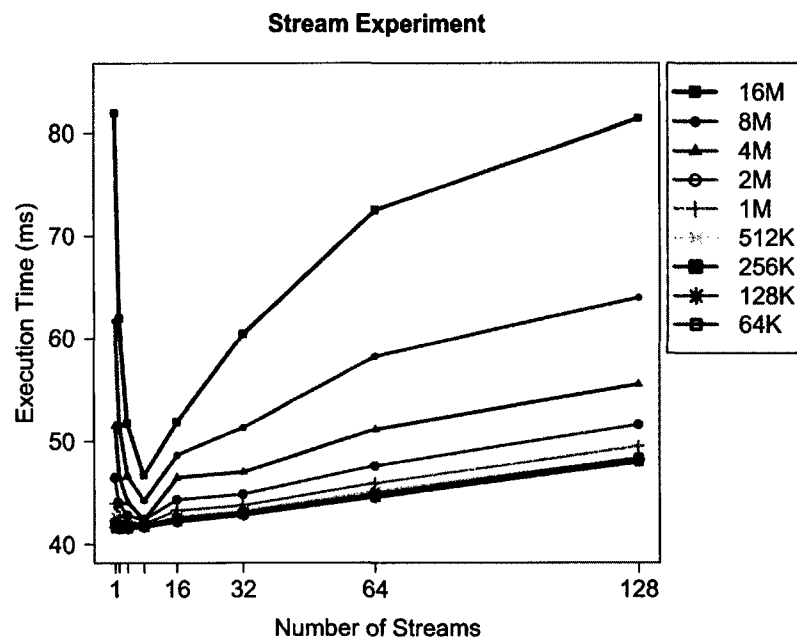


Figure 3.1: The results of simpleStreams application with various numbers of streams and array sizes

First, the experiment synchronously performs a number of actions: it operates the kernel, copies memory from device to host, and collects the elapsed time. The execution time for s is then recorded as the kernel and memory copies that are operated asynchronously. The elapsed times are then averaged over 10 repetitions. The simpleStreams has been implemented in NVIDIA CUDA SDK 2.0 [30]. The hardware environment utilized in this experiment is a GeForce 8800 GT with Intel(R) Pentium(R) D 2.80 GHz CPU.

Figure 3.1 shows that the execution times for a large data set, e.g., 16 million integers, are extremely high when these applications are run without streams. These execution times get lower when running with 2, 4, and 8 streams, respectively. These times then begin to increase again as the number of streams subsequently increases. Thus, for the purposes of this experiment, data execution time could be reduced to its minimum value when 8 streams are utilized. This value is suspected to depend on the specification of the hardware. Other parameters that affect the results are the sizes of the respective blocks and grids.

3.1.1 Performance Model of CUDA Streams

Even though streams can be utilized to improve performance as shown in the experiment in the previous section, there are other factors that affect the execution times of the GPGPU applications. The GPU memory transfer and execution times are also major factors in improving performance. In this section, the performance models for CUDA streams will be introduced in order to study the effects of those two factors.

Assuming that there are two execution blocks in a grid, hereinafter and in the following figures referred to as "block 1" and "block 2", these two blocks will be executed in parallel, and each block has to wait for the data from host-to-device memory copies. Another assumption is that the time to execute operations in each block and the time of transferring data to be executed are the same. The time diagram of non-streamed execution is shown in Figure 3.2.

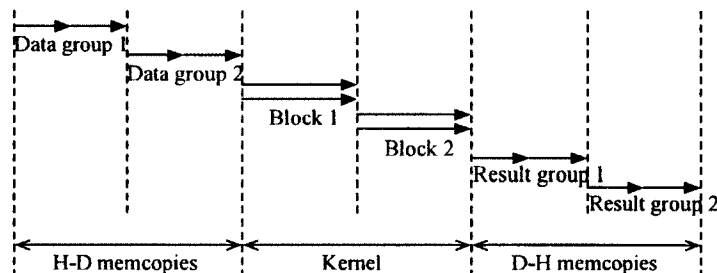


Figure 3.2: Time diagram of non-streamed execution

In Figure 3.2, the kernel waits until all data are copied from the host to the device memory and then operates on those data. Thus, the results are copied back to the host again. However, each block in the kernel may process data independently. The data can be grouped as data group 1 and data group 2, which are operated by kernel block 1 and block 2. Then, the result group 1 and result group 2 are generated, respectively. The total execution time, T_G , can be considered as stated in Equation (3.1) where T_{HD} represents the time of data transfer from the host to the device, T_K is the kernel execution time, and T_{DH} is the time of data transfer from the device to the host.

$$T_G = T_{HD} + T_K + T_{DH}. \quad (3.1)$$

Due to independent data and operations in each kernel execution, which means the operations and memory copies can be done at the same time, streams allow memory copy to overlap with kernel execution. Therefore, the total execution time of the program can be reduced. This strategy can be described by the time diagrams in Figure 3.3.

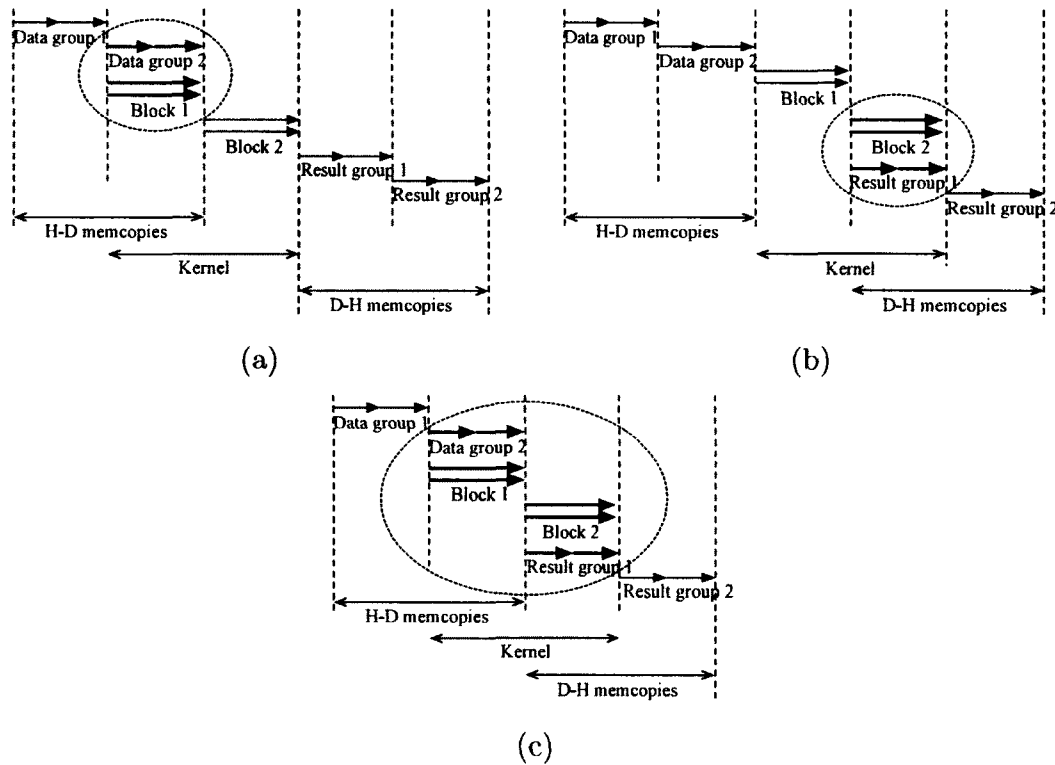


Figure 3.3: Time diagram of streamed execution: (a) asynchronous memory copies from host to device, (b) asynchronous memory copies from device to host, and (c) asynchronous memory copies in both directions.

According to the time diagram in Figure 3.3 (a), once the data group 1 transaction completes, the kernel block 1 starts operating on those data at the same time that the data group 2 is transferred to the device memory. The time of transferring data group 2 to the device memory is consequently hidden and can be

considered in Equation (3.2)

$$T_G = \frac{T_{HD}}{s} + T_K + T_{DH}. \quad (3.2)$$

Figure 3.3 (a) describes the ability of kernel and memory copies overlapping after each kernel block execution is completed. The data group 1 and group 2 are sequentially copied from the host to the device. The kernel block 1 and block 2 then operate those data, respectively. As the kernel block 1 completes its operation, the result group 1 is copied back to the host memory at the same time as the kernel block 2 is operating data group 2. The total execution time can be considered as in Equation (3.3) [30]

$$T_G = T_{HD} + T_K + \frac{T_{DH}}{s}. \quad (3.3)$$

In these two cases, the device is still idle during the time that the data are sequentially copied. Then they can be combined together for better performance as illustrated by the diagram shown in Figure 3.3 (c). It is obvious that the time of transferring data group 2 and result group 1 are hidden. Then the total execution time can be reduced as in Equation (3.4)

$$\frac{T_{HD}}{s} + T_K + \frac{T_{DH}}{s}. \quad (3.4)$$

However, the total time of execution also depends on how complicated the operations in the kernels are (kernel execution time) and dependency of operations and data. Figure 3.4 shows the time diagram of execution time when the time of memory copies are longer than kernel executions.

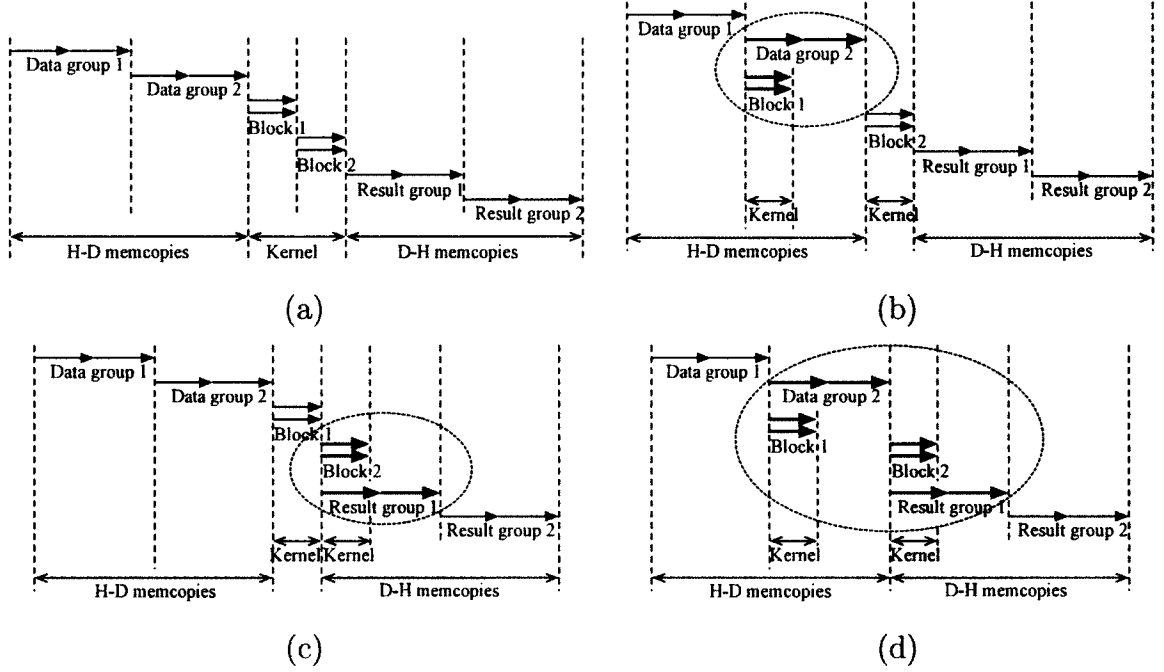


Figure 3.4: Time diagram of execution time when the time of memory copies are longer than kernel executions: (a) synchronous memory copies, (b) synchronous memory copies from host to device, (c) asynchronous memory copies from device to host, and (d) asynchronous memory copies in both directions.

Compared to the asynchronous memory copies in Figure 3.4 (a), the diagrams in Figure 3.4 (b), (c), and (d) show how the kernel execution and memory copies overlapped in the case that kernel execution times are shorter than memory copies. The total execution time for the cases shown in Figure 3.4 (b) and (c) can be considered in Equation (3.5). As shown in Figure 3.4 (d), the kernel execution time can be neglected no matter how many streams it uses. The model in this case is described in Equation (3.6)

$$T_G = T_{HD} + \frac{T_K}{s} + T_{DH} \quad (3.5)$$

$$T_G = T_{HD} + T_{DH}. \quad (3.6)$$

The time diagram in Figure 3.5 (b), (c), and (d) shows the performance of streams when the kernel execution times are longer than the time of memory copies compared to the diagram in Figure 3.5 (a). The total execution times of each diagram can be considered as in Equations (3.2), (3.3), and (3.4), respectively. In Figure 3.5 (d), if the memory copy time in a stream is much less than the kernel execution time, it supposes that there is only kernel execution time to be considered as the total execution time.

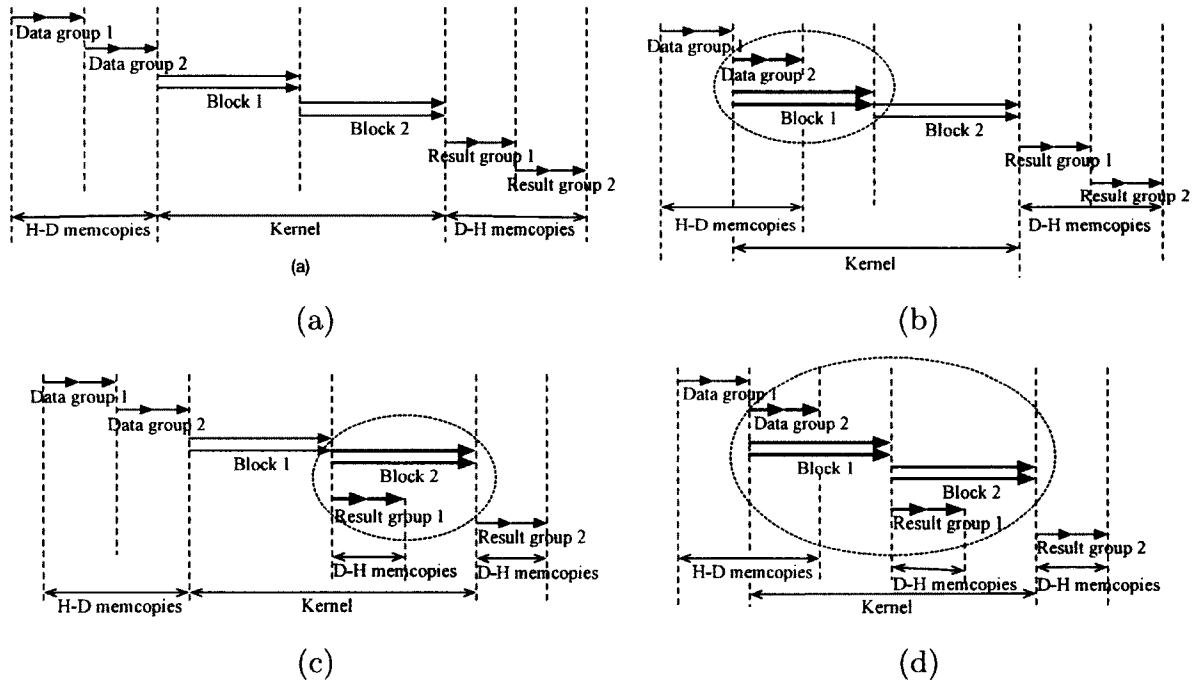


Figure 3.5: Time diagram of execution time when the time of kernel executions are longer than memory copies: (a) synchronous memory copies, (b) synchronous memory copies from host to device, (c) asynchronous memory copies from device to host, and (d) asynchronous memory copies in both directions.

3.1.2 Benchmarks

In this section, the GPGPU matrix multiplication application is introduced in order to demonstrate the utilization of streams on an application.

When two matrices A and B with dimensions of $n \times m$ and $m \times p$ are multiplied, the dimension of the result matrix C is $n \times p$. Each element of matrix C , c_{ij} , is the summation of each element in row i of matrix A , a_{ik} , multiplied by each element of column j of matrix B , b_{kj} , as shown in Equation (3.7).

$$c_{ij} = \sum_{k=1}^m a_{ik} b_{kj}. \quad (3.7)$$

To implement matrix multiplication in CUDA programming, each thread in the device computes each element of matrix C , i.e., each thread computes the summation in c_{ij} . All elements in row i of matrix A and column j of matrix B are transferred to the corresponding thread [3].

In this experiment, the multiplication of two squared matrices is considered. The size of the matrices varies from 64×64 elements to 2400×2400 elements. Both matrices are copied from the host memory to the device memory. Then the kernel multiplies those two matrices and the result matrix is copied back to the host memory. This application also runs on a GeForce 8800 GT system with Intel(R) Pentium(R) D 2.80 GHz CPU.

The operation times of matrix multiplication recorded by *CUDA Events* are shown in Table 3.1. The results suggest that the kernel times are longer as the sizes of matrices are larger. The times of transferring two matrices from the host to the device are also about twice that of transferring a result matrix from the device to the

host. Since streams allows overlapping on the GPU, we use the values in Table 3.1 with the performance model described in Section 3.1.1 to evaluate the performance increase.

Table 3.1: Times spent by operations of matrix multiplication, measured by *CUDA Events*

| Matrix | H-D memory copy (ms) | Kernel execution (ms) | D-H memory copy (ms) |
|--------------------|-------------------------|--------------------------|-------------------------|
| 64×64 | 0.16 | 0.05 | 0.05 |
| 160×160 | 0.24 | 0.15 | 0.09 |
| 320×320 | 0.64 | 0.83 | 0.28 |
| 800×800 | 3.28 | 11.76 | 1.61 |
| 1600×1600 | 12.79 | 94.45 | 6.34 |
| 2400×2400 | 28.63 | 319.35 | 14.23 |

When the sizes of matrices are 64×64 and 160×160 , the time of transferring data from the host to the device memory is longer than the kernel execution time. This case matches the time diagram in Figure 3.4 (a). Others match the time diagram in Figure 3.5 (a). The corresponding performance model for the streams applied on both host-to-device and device-to-host memory copies is described in Equation (3.6). Table 3.2 shows non-stream execution time versus 8-stream execution time. The percentage of performance increase from streams is discussed.

Table 3.2: Total execution times with non-stream and the expected total execution time with 8-stream are calculated from the operational times on Table 3.1

| Matrix | Non-stream execution (ms) | 8-stream execution (ms) | % of performance increase |
|--------------------|------------------------------|----------------------------|------------------------------|
| 64×64 | 0.26 | 0.21 | 19.23 |
| 160×160 | 0.48 | 0.33 | 31.25 |
| 320×320 | 1.76 | 0.95 | 46.02 |
| 800×800 | 16.65 | 12.37 | 25.71 |
| 1600×1600 | 113.58 | 96.84 | 14.74 |
| 2400×2400 | 362.21 | 324.71 | 10.35 |

From Table 3.2, it can be seen that when the size of matrices are small, such as 64×64 , the performance of matrix multiplication slightly increases. However, when the sizes of matrices are 320×320 , the performance of the application with 8 streams increases 46.02 percent of the application performance without stream. The performance starts to drop as the sizes of the matrices are larger because the gaps between memory transfer times and kernel execution times are extended.

Since the latency of data transfer between the host and the device is one of the major factors that impacts the performance of a GPGPU application, there are some strategies to eliminate this factor. Streaming is one of the strategies that aims to reduce the latency by overlapping kernel execution with data transfer. The proposed performance models and time diagrams show that streams can hide kernel execution time in the case that the application consumes more time on data transfer and that streams can hide memory transfer time when the device is occupied by kernel execution. However, this strategy performs well on the application when the kernel execution time and the time of memory transfer are not too much different. The results suggest that streaming strategy is effective when the application data

are independent because it can be applied on both host-to-device and device-to-host memory transfers.

3.2 Two-Level Checkpoint/Restart Protocols

This section describes the GPU checkpoint/restart (CPR) protocols with CUDA stream utilization [11] [31]. First of all, the GPU checkpoint concept is based on the following assumptions:

1. Kernel execution is sufficiently long such that the checkpoint will finish before the kernel completes. Otherwise the idea would not be beneficial.
2. The data is independent such that a transfer of data input of the next execution block can be overlapped with an execution of the current code block.
3. There is a thread in the CPU that handles the checkpoint process.

Figure 3.6 illustrates the streamed checkpoint protocol. The checkpoint cost is a result of host memory allocation and device-to-host memory copy after a thread synchronization. With CUDA streams, the kernel continues executing while the checkpoint data are copied to the host. Once the memory copy finishes, the host dumps all the checkpoint data to a checkpoint file, which is handled by the underlying CPR mechanism. When using CUDA streams, the data set is split into chunks. The first chunk is copied to the host before the next kernel is invoked. Then, the subsequent chunk is copied while the kernel is executing after the first chunk transfer completes [4].

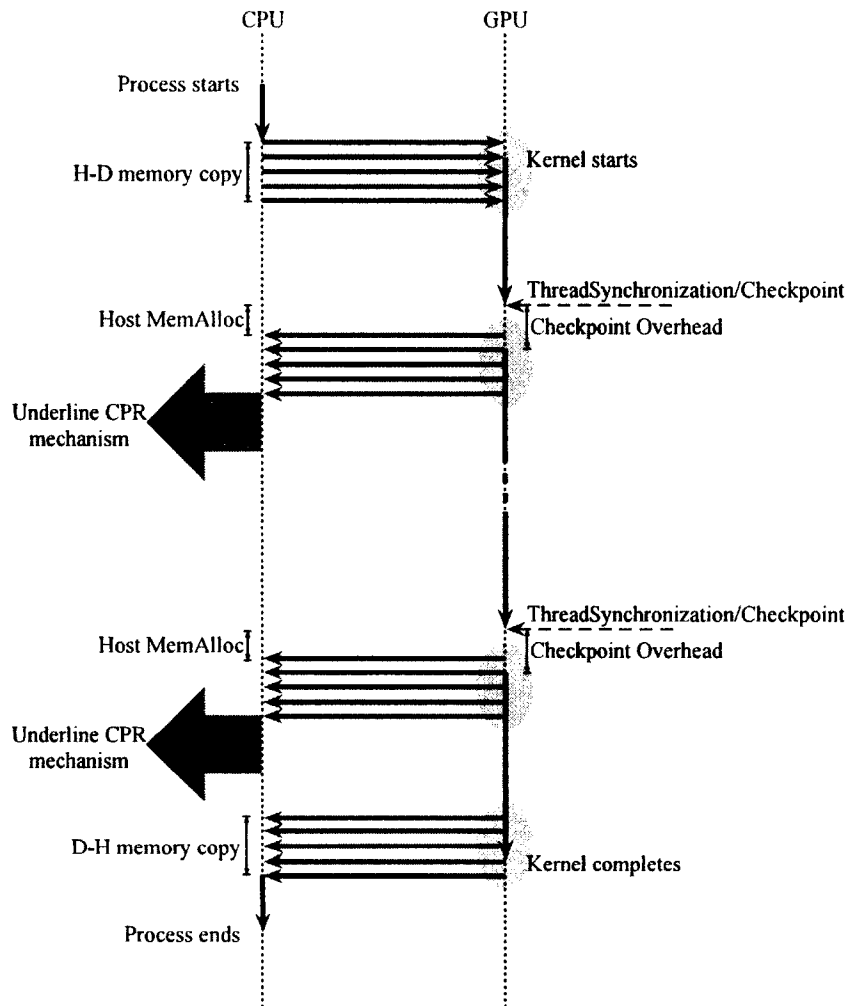


Figure 3.6: The checkpoint protocol for GPU streamed CPR

Figure 3.7 shows the streamed restart protocol. When a failure occurs while the kernel is executing, the device application context, including the device memory, is destroyed. Therefore, to restore the application, the device context has to be recreated along with device memory, and host-to-device memory has to be copied again. The restart process begins by reading the checkpoint file to host memory. However, reading the checkpoint file is handled by the underlying mechanism and the duration of this process depends on the speed of reading from the hard drive, I/O or network storage. Those factors are not considered in the GPU recovery cost. The

recovery cost includes the overhead of device memory allocation and host-to-device memory copy. CUDA streams are beneficial by starting the kernel execution while the rest of the checkpoint data is being transferred. The experiments based on the proposed CPR protocols are presented in the next section.

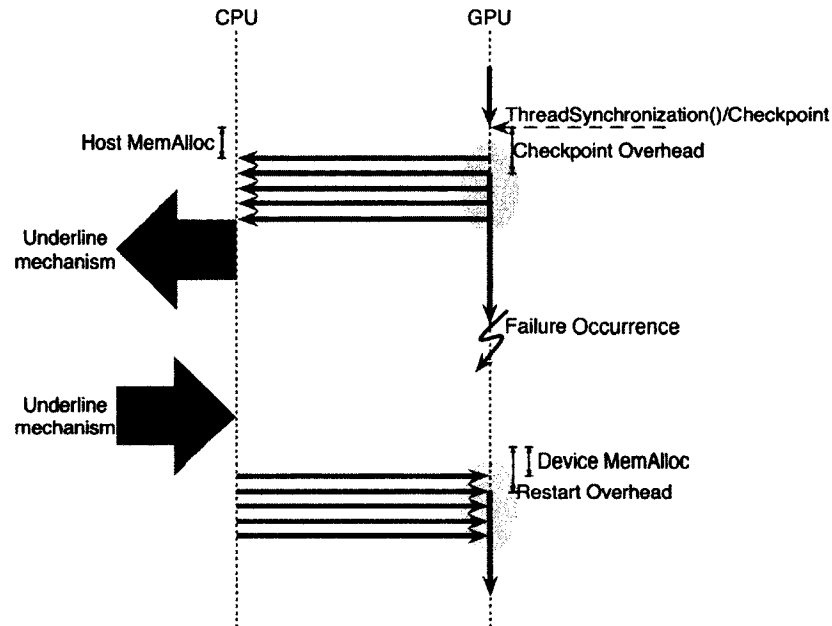


Figure 3.7: The restart protocol for GPU streamed CPR

3.3 Experiments

To study the cost of checkpoint and restart processes, the behavior of the protocols presented in the previous section is simulated and the checkpoint costs, recovery costs, and wasted time, which is the summation of checkpoint cost, recovery cost, and recomputing time due to a failure, are collected. The experiments are done on various sizes of a large array addition application with three types of GPU CPR mechanisms: non-streamed, 4-streamed, and 8-streamed CPRs.

Listing 3.1 illustrates the simulation based on the protocols in Figures 3.6 and 3.7. Since the checkpoint process takes place on the host, the kernel of the iterated array addition is split into three parts: `kernel_1()`, `kernel_2()`, and `kernel_3()`, which are prolonged by l , m , and n loops of iterative array addition ($C = A + B$), respectively. We assume that `kernel_1()` is the computation before a checkpoint, `kernel_2()` is the computation between the checkpoint and a failure, and `kernel_3()` is the computation after the failure until the application finishes. Then when a failure occurs, it has to recompute only `kernel_2()` instead of restarting from the beginning.

Listing 3.1: The pseudo code imitating GPU streamed checkpoint and restart protocols

```

1 // Begin computing
2 Do host-to-device memory copy of array A and B.
3 Execute kernel_1() with l iterations.

4 // Checkpoint process
5 Synchronize all threads to prepare for memory copy.
6 Allocate host memory for data checkpoint, i.e., for array A, B, and C.
7 Do device-to-host memory copy of array A, B, and C.

8 Execute kernel_2() with m iterations.

9 // Failure occurrence
10 Free all device memory

11 // Restart process
12 Reallocate device memory for array A, B, and C.
13 Do host-to-device memory copy of array A, B, and C.

14 // Recompute kernel_2()
15 Re-execute kernel_2() with m iterations.

16 Execute kernel_3() with n iterations.

17 Do device-to-host memory copy of the result array C.

```

The experiments are done on an NVIDIA GeForce GTX 295 system, which has compute capability 1.3. The heuristics of NVIDIA's graphic card with compute capability 1.x indicate that the maximum number of blocks in a grid is 65535, and the maximum number of threads in a block is 512 [3]. Then, the maximum number of threads is 65535×512 , or $2^{25} - 2^9$. For the purpose of load balancing and data correctness while invoking the kernel with streams, we vary the size of arrays from 2^{10} to 2^{24} . If the size of arrays reaches 2^{25} , the number of iterations in the kernel will be influenced.

In a non-streamed case, the memory copy and the kernel execution are done consecutively. However, in 4-streamed and 8-streamed cases, those instructions are done simultaneously. The checkpoint cost is obtained by timing host memory allocation and device-to-host memory copy. The recovery cost is obtained by timing device memory reallocation and host-to-device memory copy. The wasted time is obtained by the summation of those costs and the time of `kernel_2()` execution. The results are shown in Figure 3.8.

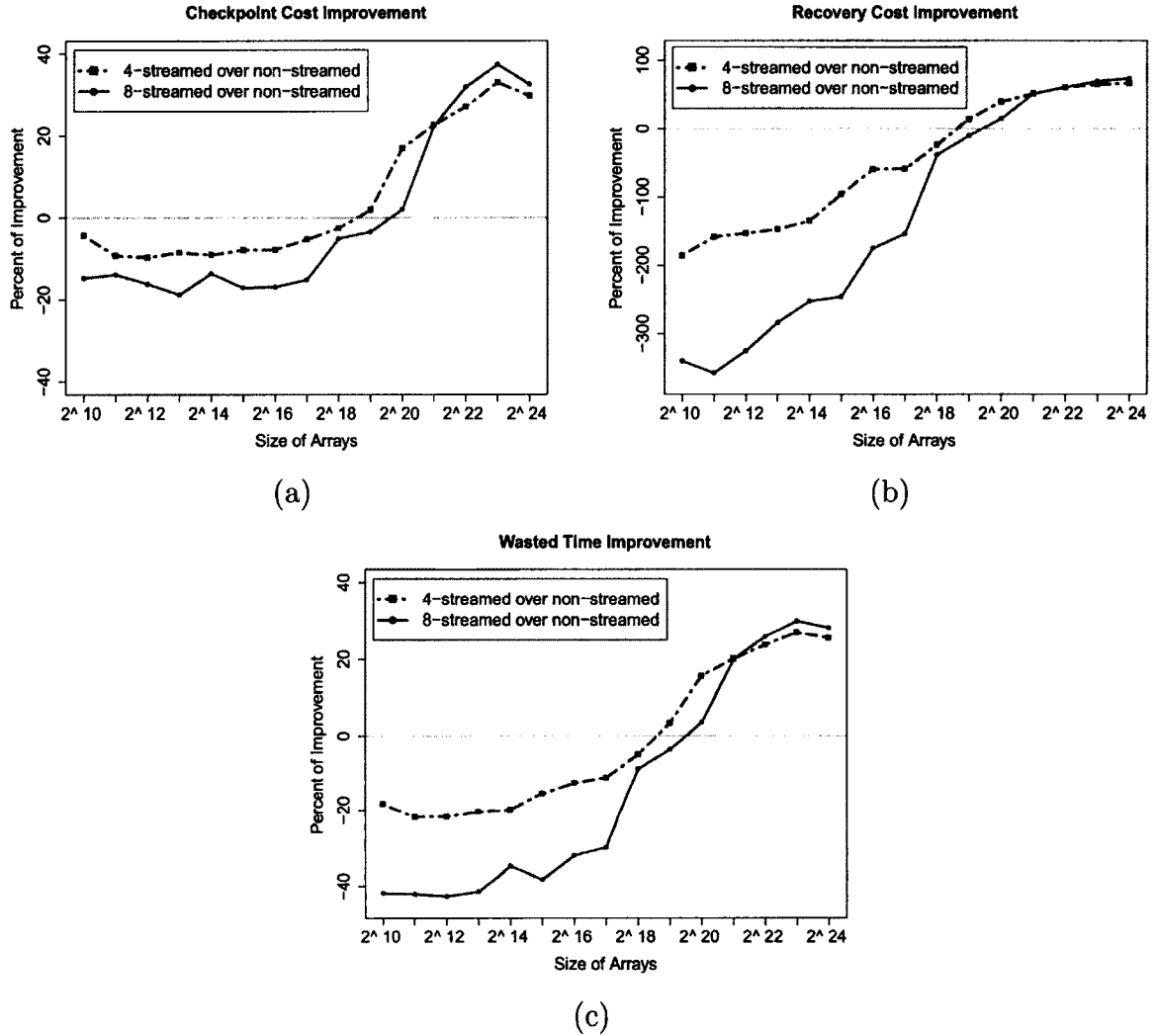


Figure 3.8: The percentage of performance improvement in terms of (a) checkpoint cost, (b) recovery cost due to a failure occurrence, and (c) wasted time due to a failure occurrence

Figure 3.8 (a) illustrates percentage of the performance improvement of streamed CPR in terms of checkpoint cost. When the size of arrays is less than 2^{19} , the percentage of improvement is negative since the streamed CPR has no advantage over non-streamed CPR. However, it becomes positive when the size of arrays is 2^{19} . When the size of arrays is 2^{24} , 4-streamed and 8-streamed CPRs gain advantage over non-streamed CPR with the percentages of 29.790 and 32.699, respectively.

Similar to the checkpoint cost, as shown in Figure 3.8 (b), the recovery cost of 4-streamed CPR is smaller than non-streamed CPR when the size of arrays is at least 2^{19} . Also, the recovery cost of 8-streamed CPR gains advantage over 4-streamed CPR when the size of arrays is at least 2^{21} . As the size of arrays is 2^{24} , the percentages of improvement of 4-streamed and 8-streamed CPR are 66.743 and 74.305, respectively. This similarity is because both checkpoint and recovery costs depend mainly on the duration of memory copy.

The wasted times due to a failure occurrence on an application with non-streamed, 4-streamed, and 8-streamed CPRs are illustrated by Figure 3.8 (c). Since the wasted time is the aggregate of checkpoint cost, recovery cost, and recomputing time, the performance improvement in the aspect of wasted time is similar to the checkpoint and recovery costs.

In this experiment, the benefits of the streamed CPR mechanism with only one checkpoint and one failure is studied. To study the benefits of streamed CPR on real-world applications, the simulation on a long-run application and its results will be presented in the next section.

3.4 Simulation

In the simulation to study the benefits of streamed CPR on a long-run application, despite various checkpoint intervals, the mean-time-to-failures (MTTFs) must be considered. Due to the fact that a large scale GPU cluster system, such as ORNL's Titan, LLNL's Sequoia, etc. [1] is a heterogeneous system with a significant number of GPUs, the MTTF of the system depends on the MTTF of other modules

or nodes in the system. In this study, the MTTFs are varied from 12 hours to 7 days with the checkpoint interval of 30 and 120 minutes. The application length is fixed at 1000 hours. Since both 4-streamed and 8-streamed CPR have an advantage over non-streamed CPR when the size of arrays is over 2^{19} , the simulation is done for the array size of 2^{24} . The performance is observed in three different aspects: (1) the percentage of total checkpoint costs compared to wasted time; (2) the percentage of total recovery costs compared to wasted time; and (3) the percentage of wasted time compared to the completion time, which is the summation of the application length and the wasted time. Figure 3.9 illustrates the percentages of total checkpoint costs, while Figures 3.10 illustrates the percentages of total recovery costs, and Figures 3.11 shows the percentages of wasted time.

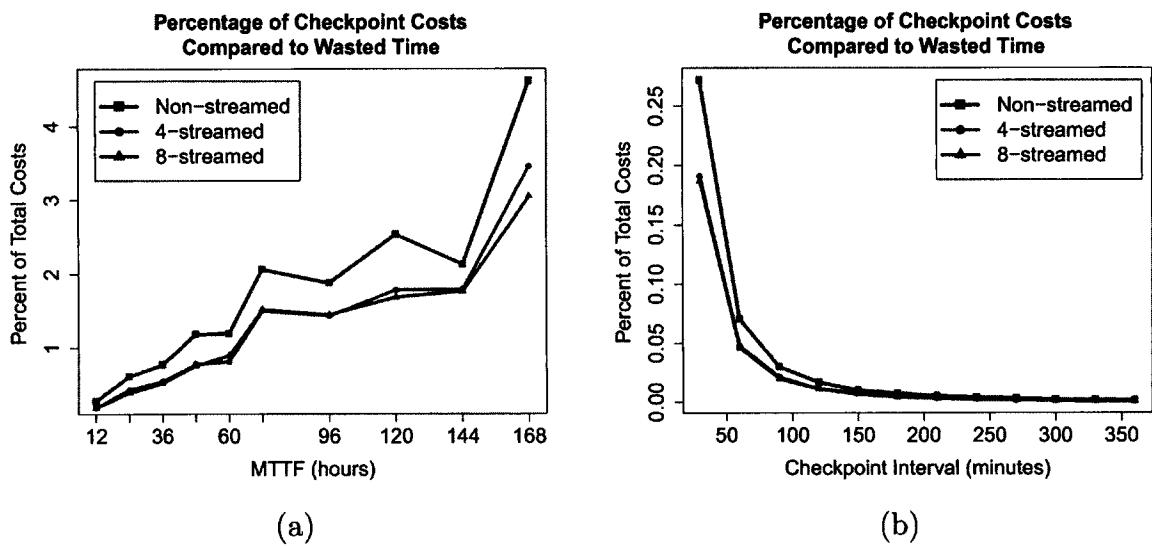


Figure 3.9: The percentages of total checkpoint costs compared to wasted time of the application with non-streamed, 4-streamed, and 8-streamed CPRs, when the size of arrays is 2^{24} and (a) the checkpoint interval is 30 minutes, and (b) the MTTF is 12 hours.

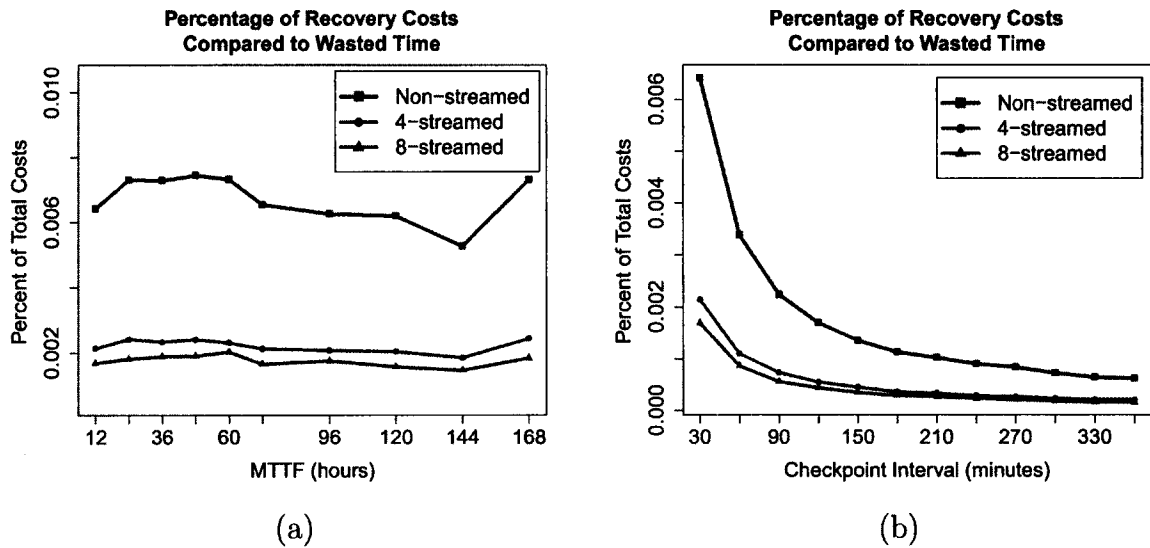


Figure 3.10: The percentages of total recovery costs compared to wasted time of the application with non-streamed, 4-streamed, and 8-streamed CPRs, when the size of arrays is 2^{24} and (a) the checkpoint interval is 30 minutes, and (b) the MTTF is 12 hours.

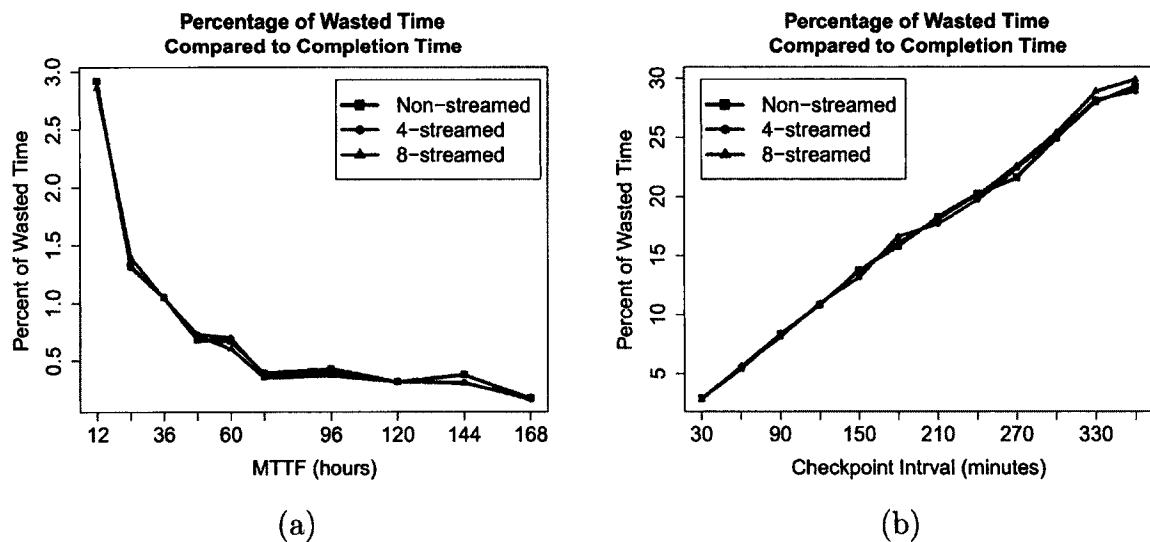


Figure 3.11: The percentages of wasted time compared to completion time of the application with non-streamed, 4-streamed, and 8-streamed CPRs, when the size of arrays is 2^{24} and (a) the checkpoint interval is 30 minutes, and (b) the MTTF is 12 hours.

Figure 3.9 illustrates the percentages of total checkpoint costs compared to wasted time when the size of arrays is 2^{24} . According to Figure 3.8 (a), when the size of arrays is 2^{24} , the checkpoint cost of 8-streamed CPR is the smallest. As a result, 8-streamed CPR performs better than non-streamed and 4-streamed CPRs in term of total checkpoint overheads, particularly when there are more checkpoints. In addition, the CPRs with the checkpoint interval of 30 minutes, shown in Figure 3.9 (a), produce larger total checkpoint costs when the MTTF increases. Moreover, with various checkpoint intervals, shown in Figure 3.9 (b), the total checkpoint overheads are smaller for the larger checkpoint interval. As non-streamed CPR generates the most checkpoint costs in both cases, the total checkpoint costs are very small compared to the wasted time with no more than 5 percent.

Figure 3.10 illustrates the percentages of total recovery costs compared to wasted time when the size of arrays is 2^{24} . Since the number of failures depends on MTTF, when the MTTF increases, both recovery costs and wasted time decrease. As a result, the graphs slightly change. That is between 0.006 – 0.008 percent for the non-streamed CPR, 0.0020 – 0.0025 percent for the 4-streamed CPR, and 0.0015 – 0.0020 percent for the 8-streamed CPR (as shown in Figure 3.10 (a)). On the other hand, when the MTTF is fixed at 12 hours, by varying the checkpoint intervals (as shown in Figure 3.10 (b)), the graphs drop due to the increase of checkpoint costs, which makes the wasted time increase. Furthermore, in both cases, streamed CPR obviously performs better than non-streamed CPR. However, the recovery costs do not have much effect on the performance of the application since the percentages of total restart overheads are less than 0.01 percent of wasted time in any cases.

Figure 3.11 illustrates the percentage of wasted time compared to completion time, which is the summation of wasted time and application length. When the checkpoint interval is fixed at 30 minutes (shown in Figure 3.11 (a)), as the MTTF increases, the percentage of wasted time tends to be smaller. However, when the MTTF is fixed at 12 hours (shown in Figure 3.11 (b)), as the checkpoint interval increases, the percentage of wasted time also increases because the recomputing time increases. Nevertheless, the costs of non-streamed, 4-streamed, and 8-streamed CPRs are relatively small compared to the recomputing time and completion time. Thus the difference of wasted time between those three types of CPRs is insignificant.

3.5 Conclusion

Even though the checkpoint/restart mechanism can improve the application resilience, it reduces the performance by the cost of the checkpoint process. In this study, the GPU checkpoint/restart protocol that aims to reduce the fault tolerance cost is proposed. The experiments have revealed that the streamed CPR can reduce the checkpoint cost when the size of checkpoint data is large enough. Additionally, it can improve the restart process by reducing the recovery cost.

The simulation has shown that the streamed CPR can reduce the cost of the checkpoint process. However, in a long-run application, since the costs of both checkpoint and restart processes are relatively small compared to the recomputing time and the completion time, streamed CPR may not be beneficial on a single node GPU system.

CHAPTER 4

PERFORMANCE MODEL FOR GPGPU

Although a GPU is considered an accelerator for a CPU, the architectures of a GPU and a CPU are quite different [3] [32]. In order to estimate an application completion-time, the kernel execution time on a GPU has to be estimated. Thus, a programmer can make a decision whether the GPU is worth participating in the computation and can further improve the heterogeneous computing application. In addition, the performance model can also help to increase the efficiency of fault tolerance techniques for a GPGPU application, such as checkpoint/restart mechanisms, especially for a large GPU cluster. One of the major problems in this area is checkpoint scheduling. To find an optimum checkpoint placement, the time-to-failure (TTF) of the system and the completion-time of the application must be taken into account. The system TTF can be obtained by a failure prediction technique, while the completion time of the application can be estimated by the performance model.

In this study, a novel performance model is proposed. The model based on the current state-of-the-art model by Hong and Kim [16]. However, Hong and Kim's model is limited by an assumption that a latency due to every global memory instruction is the same. Nonetheless, the empirical results reveal that the memory latencies are varied depending on the data type and the type of memory access.

Figure 4.1 shows that a global memory write operation takes much longer than a global memory read operation in every case. Also, 4-byte word accesses take longer than 1-byte word accesses. Thus it is important to consider various memory access costs that will impact GPU performance outcome.

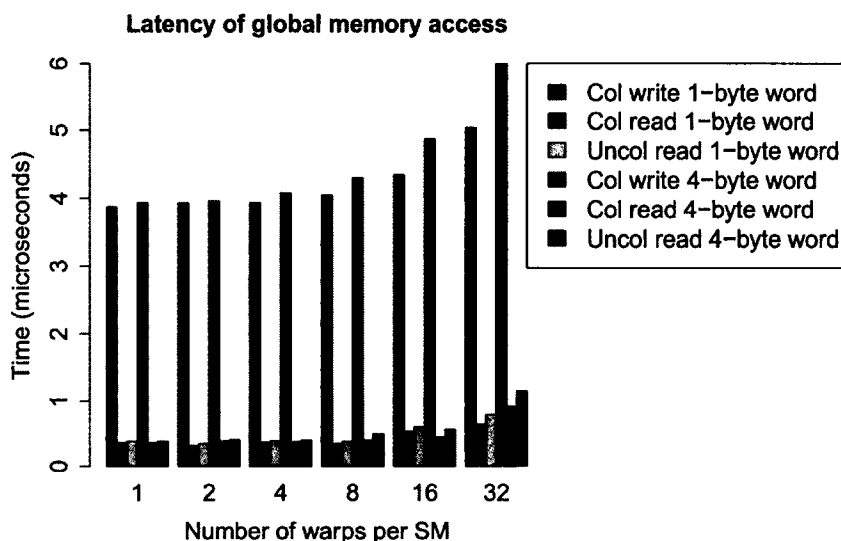


Figure 4.1: Memory latency on GPU NVIDIA GeForce GTX 295

Since the detailed architecture of a Graphic Double Data Rate (GDDR) memory is not public, the actual memory latencies are measured. Then instead of using fixed memory latencies like in Hong and Kim’s model, the latencies from an actual benchmark are used in our model as described in Section 4.1.1.

Moreover, Hong *et al* [16] considers a latency for a shared memory instruction to be the same as a computation instruction. In fact, a computation instruction usually takes approximately 20 clock cycles while a shared memory takes 38 cycles according to [20], and 40 cycles according to the experiment.

Due to the aforementioned reasons, our model does not rely on *PTX* code – an assembly-like code that is generated by compiling with *nvcc* – to obtain the number of registers, the number of computations, and memory instructions [3] [33]. On the other hand, those instructions can be counted from CUDA code.

4.1 Performance Modelling

In this section, the important parameters in the proposed performance model are defined. Then, the model notations in a general case and in cases with synchronization functions are described. Furthermore, the impacts of *branch divergence* and *bank conflicts* are also discussed in detail.

4.1.1 Parameters

In GPGPU programming, parallel instructions are executed by multiple threads. These threads are maintained as thread *blocks* that will be assigned to stream processors (SMs) as illustrated in Figure 4.2(a). The thread blocks are also organized into a *grid* of a kernel. Moreover, Figure 4.2(b) shows that threads in a block executed in an SM are managed in a group of parallel threads called a *warp*.

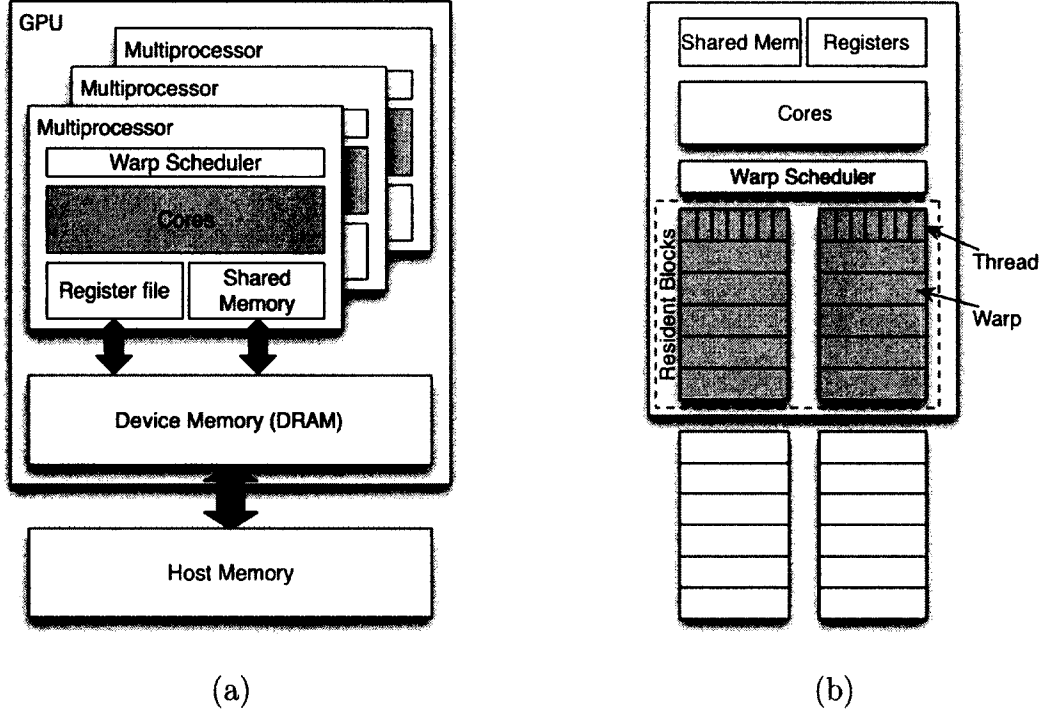


Figure 4.2: GPU architecture: (a) programming model; (b) warp scheduling in an SM

Let S_G be the number of blocks in a grid (so called grid size), and the number of SMs be N_{SM} . Then the number of blocks assigned to an SM is defined by

$$N_B = \left\lceil \frac{S_G}{N_{SM}} \right\rceil. \quad (4.1)$$

However, not all assigned blocks can be resided into an SM. The new blocks will be assigned to an SM as it completes the execution of the previous blocks [3] [34]. Let N_{RB} be the number of blocks that can be resided into an SM, which is called resident blocks, and N_P be the number of groups of blocks assigned to a multiprocessor. Then, N_P can be defined as

$$N_P = \left\lceil \frac{N_B}{N_{RB}} \right\rceil. \quad (4.2)$$

Moreover, N_{RB} is restricted by hardware limitation, i.e., the maximum number of resident blocks ($N_{RB,max}$), the maximum number of resident warps ($N_{RW,max}$), and the limitation of resources, such as the amount of shared memory and registers [3] [34]. Let each block assigned to an SM have S_B threads. Those threads are grouped into warps, where each warp has S_W threads. Then N_{RB} is defined by

$$N_{RB} = \min \left\{ N_B, N_{RB,max}, \left\lceil \frac{N_{RW,max} S_W}{S_B} \right\rceil, \left\lceil \frac{M_{S,max}}{M_{SB}} \right\rceil, \left\lceil \frac{R_{max}}{R_B} \right\rceil \right\}, \quad (4.3)$$

where $M_{S,max}$ and R_{max} are the maximum amount of shared memory (in bytes) and the maximum number of 32-bit registers per SM, respectively. These values are hardware specifications while M_{SB} and R_B are the amount of shared memory and the number of registers required by a block, respectively. Both M_{SB} and R_B can be estimated mathematically as described in the NVIDIA Programming Guide [3].

Furthermore, the number of resident warps is defined by

$$N_{RW} = \left\lceil \frac{N_{RB} S_B}{S_W} \right\rceil. \quad (4.4)$$

Note that there is a limit to the number of threads per block and to the number of blocks per grid. That is, S_G and S_B cannot exceed the hardware heuristics.

Furthermore, there are 4 additional parameters to be introduced in this section:

(1) the operation time required by the total set of computational operations, T_C ; (2) the operation time due to the last set of computational operations, T_{CL} ; (3) the memory latency resulted from the total set of memory instructions, T_M ; and (4) the memory latency lost by the last memory instruction in the kernel, T_{ML} .

From Figure 4.3, suppose that there are 2 resident blocks assigned to an SM, each resident block has 2 warps, and there are 5 instructions to be executed by each

thread. Since I1, I3, and I4 are computational instructions, T_C is a summation of the operation times due to I1, I3, and I4, where T_{CL} is an aggregation of the operation times due to I3, and I4. Also, T_M is the memory latency resulted by I2 and I5, while T_{ML} is a result of memory access in I5 only.

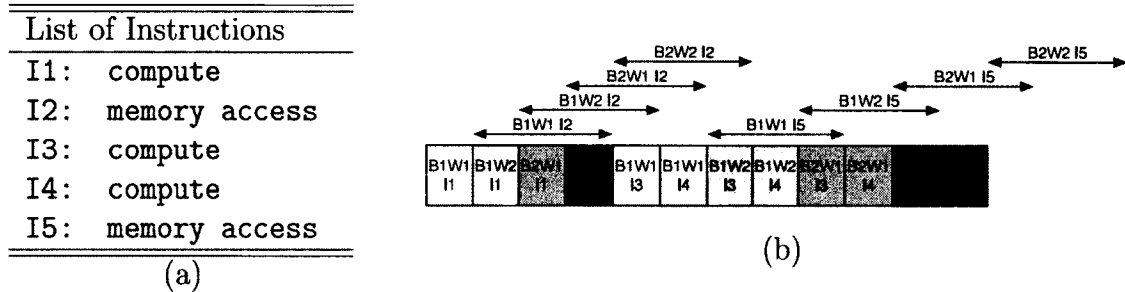


Figure 4.3: The instruction list in (a) transforms into the timeline in (b).

Since $T = C \times F_P$, where T is the operation time, C is the number of clock cycles required by an operation, and F_P is the frequency of an SM, the number of clock cycles has to be determined in order to evaluate the operation time or the memory latency. The number of clock cycles due to the set of computational operations can be analyzed and derived from the kernel code, which is detailed in the NVIDIA Programming Guide [3].

Nonetheless, evaluating the memory latency is more complex. For a global memory access, the global memory latency is varied depending on the size of data, memory bandwidth, and the number of transactions required by a memory instruction (memory coalescing). According to the NVIDIA Programming Guide [3], a global memory latency can be as high as 400 to 800 clock cycles per memory access.

The CUDA C Best Practices Guide [32] also states that the latency for reading an uncached data from local or global memory ranges from 400 to 600 clock cycles.

For these reasons, instead of using a fixed number of clock cycles, the operation time from the measurement divided by N_P is considered in order to estimate the global memory latency.

4.1.2 Performance Model in a General Case

In this section, the performance model is described by assuming that there is no branch convergence in a warp. Thus, every thread in a warp executes the same set of instructions. Then the computation times and memory latencies are the same for each warp. Moreover, assuming that there is no synchronization instruction, the idle time does not depend on the time that a warp waits for other warps to complete the instructions prior to the barrier, but depends only on memory access.

Figure 4.4(a) illustrates the case that T_C is longer than T_M , or N_{RW} is large enough to hide the idle time. The execution time of the kernel $T(K)$ can be defined as

$$T(K) = (T_{ML} + T_C N_{RW}) N_P. \quad (4.5)$$

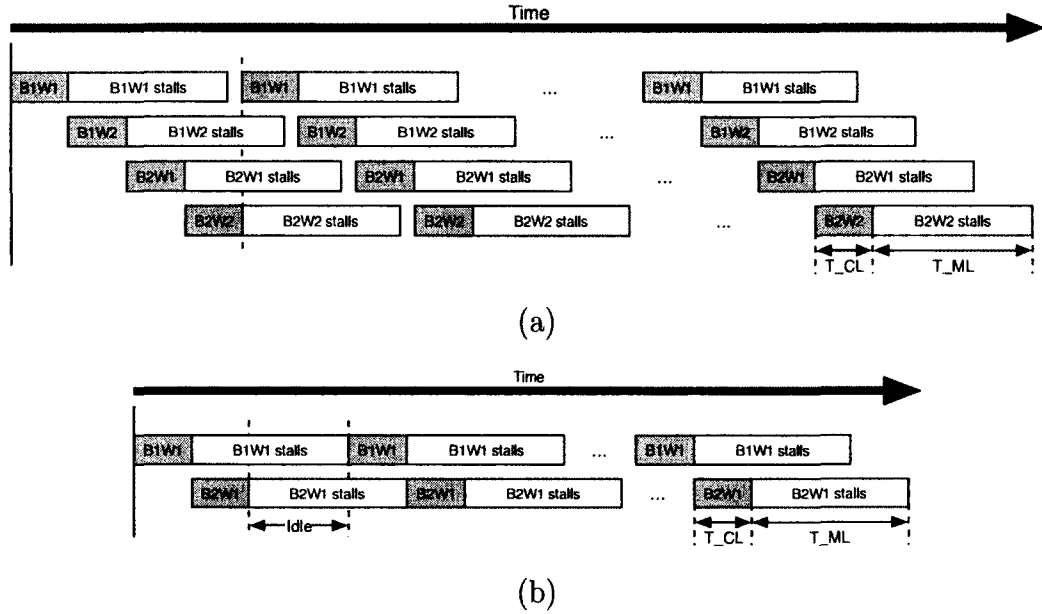


Figure 4.4: Diagrams show the execution time when: (a) there is no idle time; (b) there is idle time. A shaded box indicates that the warp is being executed. A white box indicates that the warp is waiting for the memory access.

However, in Figure 4.4(b), N_{RW} is not large enough to fill the idle time [3] [32].

Hence,

$$T(K) = (T_M + T_C + T_{CL} (N_{RW} - 1)) N_P. \quad (4.6)$$

To determine whether N_{RW} is large enough to hide the idle time, Equation (4.5) is subtracted from Equation (4.6) to find the idle time and set to zero. Consequently, $N_{RW} = \frac{T_M - T_{ML}}{T_C - T_{CL}} + 1$. Therefore, if $N_{RW} \geq \left\lceil \frac{T_M - T_{ML}}{T_C - T_{CL}} \right\rceil + 1$, there will not be an idle time.

4.1.3 Performance Model in a Case with Synchronization

In this section, a model that considers synchronization in a kernel is described. It is an extension from the model for a general case presented in the previous section.

A synchronization function or `__syncthread()` acts as a barrier at which threads in a block must wait until every thread hits this point before they are allowed to execute the next instruction. However, threads from different blocks do not have to wait [3]. In general, if there are multiple resident blocks scheduled in an SM ($N_{RB} > 1$), the compiler will try to fill the idle clock cycles with warps from the other blocks. As a result, the impact of synchronization functions can be omitted. Nonetheless, if there is only one resident block in an SM ($N_{RB} = 1$), the amount of execution time will be increased by the waiting time resulted by the synchronization function. The time diagrams in Figure 4.5 describe impacts of synchronization functions.

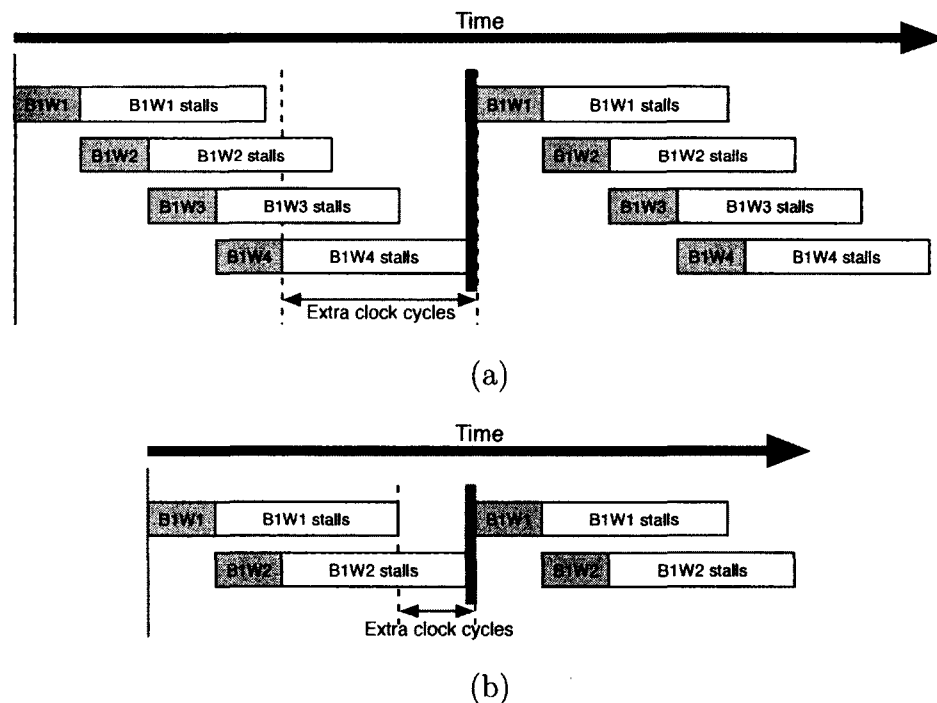


Figure 4.5: Diagrams show the execution time when: (a) a synchronization function causes extra time between two consecutive computational instructions; (b) a synchronization function causes extra time between memory and computational instructions. Again, a shaded box indicates that the warp is being executed. A white box indicates that the warp is waiting for the memory access.

Figure 4.5(a) is the diagram extended from Figure 4.4(a), which is the case that it originally does not have idle time. In this case, the synchronization function generates an extra time, $T_{Syn} + T_{M,Syn}$, where T_{Syn} is the operation time for executing the synchronization function, and $T_{M,Syn}$ is the memory latency due to the memory instructions before or after the synchronization function. Let the number of synchronization functions be N_{Syn} . Therefore, the extra time caused by synchronization functions for each block is $(T_{Syn} + T_{M,Syn}) N_{Syn}$. Then the kernel execution time defined by Equation (4.5) can be improved as follows

$$T(K) = [T_{ML} + T_C N_{RW} + (T_{Syn} + T_{M,Syn}) N_{Syn}] N_P. \quad (4.7)$$

Figure 4.5(b) is the diagram modified from Figure 4.4(b). It illustrates the extra time due to a synchronization function in the case that it originally has idle time. The extra time generated by the synchronization function is $T_{Syn} + T_{C,Syn} (N_{RW} - 1)$, where $T_{C,Syn}$ is the time due to the set of computational instructions between the memory instructions and the synchronization function. Therefore, the kernel execution time defined by Equation (4.6) can be improved as

$$T(K) = [T_M + T_C + T_{CL} (N_{RW} - 1) + (T_{Syn} + T_{C,Syn} (N_{RW} - 1)) N_{Syn}] N_P. \quad (4.8)$$

4.1.4 Control Flow Cases

In kernels, control flow statements, such as **if**, **else**, **switch**, **for**, **do**, and **while**, may cause *branch divergence* in a warp, meaning that threads in a warp execute different code paths. Once threads in a warp hit a diverging point, the warp sequentially executes each branch path taken by disabling threads that are not on that path. When all paths complete, the threads re-converge back to the same

execution path [3] [35]. Consequently, branch divergence prolongs the execution time by increasing the number of instructions.

Listing 4.1: An example of pseudo code that results in *branch divergence*

```
1 tid = threadIdx.x;  
2 if tid is an even number then  
3   operation A;  
4 else  
5   operation B;  
6 end
```

For instance, the code in Listing 4.1 describes that half of the threads in a warp do operation A, another half do operation B. Suppose that there are 8 threads in a warp, once they hit the if statement, t1, t3, t5, and t7 have to wait until t0, t2, t4, and t4 finish operation A. Then t1, t3, t5, and t7 will do operation B. If operations A and B take 4 clock cycles each, the warp will take 8 clock cycles to execute the operations in the if - else statement.

Another possible issue that may occur when using shared memory is *bank conflicts*. Since shared memory in an SM is separated into a certain number of banks, there is a possibility that two or more threads may access different data in the same bank and cause threads to sequentially access that bank. Therefore, the time to access shared memory is prolonged depending on the hardware architecture.

4.2 Results and Discussion

In this section, the code analysis using the example of CUDA code is described. Additionally, naïve and tiled matrix multiplications are used as benchmarks to validate the proposed models. The GPU that is used in this experiment is NVIDIA GeForce GTX 295.

4.2.1 Code Analysis

To illustrate how we obtain the parameters introduced in Section 4.1, the kernel code of a matrix multiplication listed in Listing 4.2 is analyzed. Suppose that the programmer specifies the number of blocks and the number of threads in a block as equal to the dimension of square matrices to be multiplied (dim), which in this example is 128. Then, $S_G = S_B = 128$. The values of other parameters are summarized in Table 4.1. Since the shared memory is not used in this kernel, $M_{S,max}$ and M_{SB} in Equation (4.5) are omitted.

Listing 4.2: The kernel code of simple matrix multiplication in CUDA

```

1 int i = blockIdx.x;
2 int j = threadIdx.x;
3 int dim = blockDim.x;

4 float Cvalue = 0.0;
5 for int k = 0; k < dim; k ++ do
6     Cvalue += A[i*dim+k] * B[k*dim+j];
7 end
8 C[i*dim+j] = Cvalue;

```

Table 4.1: The values of experimental parameters validated in the model

| Parameters | Descriptions |
|---|--|
| $N_{SM} = 30$ | Hardware specification |
| $N_B = \lceil 128/30 \rceil = 5$ | Equation (4.1) |
| $N_{RB,max} = 8$ | Hardware specification |
| $N_{RW,max} = 1024$ | Hardware specification |
| $S_W = 32$ | Hardware specification |
| $R_{max} = 16384$ | Hardware specification |
| $R_B = 1024$ | There are 10 registers required by this kernel. The CUDA C Programming Guide [3] also shows how to determine R_B . |
| $N_{RB} = \min\{5, 8, \lceil 1024 \times 32/128 \rceil, \lceil 16384/1024 \rceil\}$ | Equation (4.3) |
| $N_P = \lceil 5/5 \rceil$ | Equation (4.2) |
| $N_{RW} = \lceil 5 \times 128/32 \rceil = 20$ | Equation (4.4) |
| $T_C = 14.5894$ | Code analysis |
| $T_{CL} = 0.0644$ | Code analysis |
| $T_M = 121.0543$ | Code analysis |
| $T_{ML} = 4.9199$ | Code analysis |

From the pseudo code in Listing 4.2, one can see that each thread declares five parameters and calculates the position of matrix A and matrix B. Also, when the thread executes for-loop, there are seven computations in each iteration. Then, the thread computes the position of matrix C that takes three more computations. Since each computation takes 20 clock cycles, and the frequency of NVIDIA GTX 295 is 1242 MHz, $T_C = (7 + 128 \times 7 + 3) \times 20/1242 = 14.5894 \mu s$, and $T_{CL} = 4 \times 20/1242 = 0.0644 \mu s$.

Furthermore, in each iteration of the kernel computation, there are two global memory accesses: one is reading elements from matrix A, which is coalesced, while the other is reading elements from matrix B, which is uncoalesced. From our experiment, a coalesced memory read takes $0.4289 \mu s$ and an uncoalesced memory read takes

0.4784 μs . Finally, each thread writes a value of matrix C to the global memory, which takes 4.9199 μs . Therefore, $T_M = (0.4289 + 0.4784) \times 128 + 4.9199 = 121.0543$ μs , and $T_{ML} = 4.9199$ μs .

From the experimental parameters in Table 4.1, one can determine whether there are idle times by $\lfloor (T_M - T_{ML}) / (T_C - T_{CL}) \rfloor + 1 = 70$, which is more than the number of resident warps, N_{RW} . Hence there are not enough resident warps to hide the idle time. Thus $T(K) = (121.0543 + 14.5894 + 0.0644(20 - 1)) \times 1 = 138.8673$ μs .

4.2.2 Results

We use naïve and tiled matrix multiplications as benchmarks to validate our model. The sizes of squared matrices vary from 32×32 to 512×512 . The results are shown by the graphs in Figures 4.6 and 4.7.

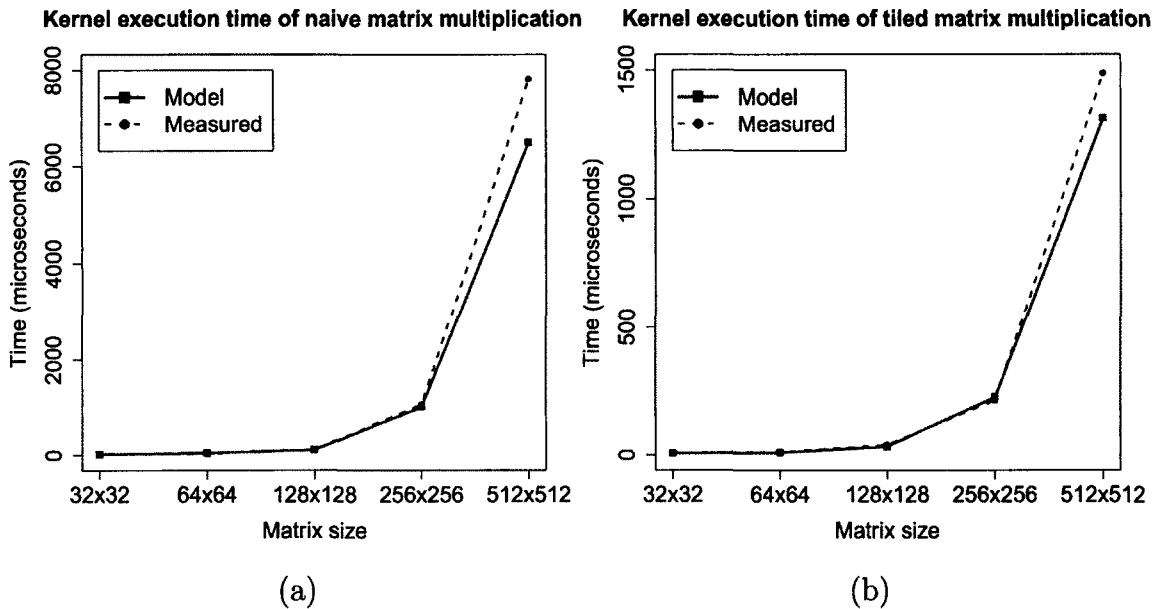


Figure 4.6: The kernel execution time from the performance model compared to the measurement of: (a) naïve matrix multiplication; (b) tiled matrix multiplication

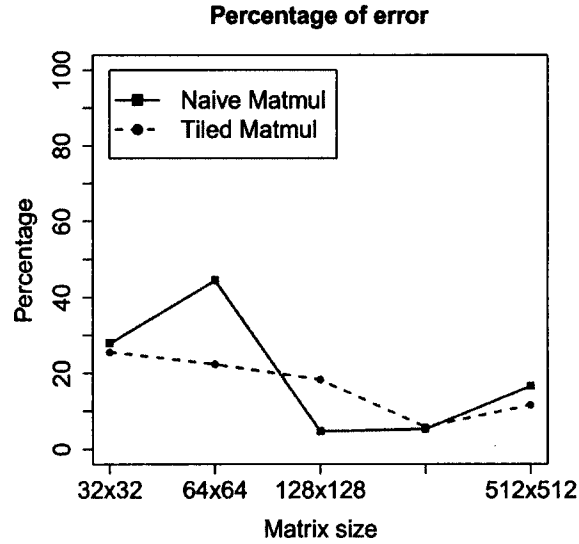


Figure 4.7: Percentage of error of the performance model compared to the measurement for both benchmarks

Figure 4.6(a) illustrates the kernel execution time of the naïve matrix multiplication estimated by the proposed performance model compared to the measurement. For the matrix size of 32×32 , the execution time from the model is only 0.005 millisecond different from the measurement, and 0.015 millisecond different for the matrix size of 64×64 , even though the graph in Figure 4.7 indicates that the percentage of error is relatively high. This is because the execution times for both cases are very small. For the matrix size of 512×512 , the execution time from the model is 1.3 ms different from the measurement, or approximately 16% error.

Figure 4.6(b) illustrates the kernel execution time of the tiled matrix multiplication estimated by the model compared to the measurement. Again, the analytical execution time from the model differs from the measurement by 0.002 millisecond for the matrix size of 32×32 , and 0.003 milliseconds for the matrix size of 64×64 . For

the matrix size of 512×512 , it differs from the measurement by 0.175 millisecond, or approximately 11% error.

Both Figures 4.6(a) and 4.6(b) also indicate that the execution time from the model follows the measurement well. Figure 4.7 shows that, for both benchmarks, our model is more accurate when the size of the matrix is practically large enough.

In conclusion, the proposed performance model shows improvements of the kernel execution time estimation for a GPGPU application, which considers memory access types and the impacts of other important factors such as synchronization functions, branch divergence, and bank conflicts. It has also suggested that the global memory latency is varied depending on the type of memory access and the data type. The results have also indicated that our performance model is more accurate when the data size is practically large enough.

CHAPTER 5

GPU APPLICATION PERFORMANCE VS CHECKPOINTS

Since a checkpoint restart mechanism can sustain application execution and performance by reducing the recomputing time when failures occur [11], [31], the purpose of this work is to optimize fault tolerance costs, while balancing the cost factors and the application performance. We propose mathematical models to address the following:

1. An optimal number of nodes for maximum system reliability and the desired application performance.
2. Near optimal checkpoint placements that mitigate system failures but still maintain the desired performance level.

In previous work [11], [31], a checkpoint/restart protocol that aims to reduce the GPU checkpoint cost and rollback using a latency hiding strategy has been proposed. Additionally, it has been argued, [31], that only cost reduction is not enough to minimize the wasted time due to a system failure. This is because the wasted time is dominated by the recomputing time.

There are existing checkpoint scheduling models that aim to minimize wasted time. Liu *et al* [12] proposed a scheme to derive a sequence of checkpoint placements

that uses the theory of a stochastic renewal reward process. Although their model targets a large-scale HPC system, it can be applied to any system as long as the system reliability is derivable.

The study here is based on the reliability models presented by Gottumukkala and Thanakornworakij [14] [15]. However, the k -node system is extended such that each node has a co-processor, for instance, a GPU. In addition, the proposed checkpoint scheduling model is an enhancement of Liu *et al* [12] and Nichamon *et al* [13]'s models in the sense that the system is a heterogeneous system and the model relies on the two-step checkpoint/restart protocol.

5.1 Application Performance and System Reliability

Since the system to be considered is a GPU cluster, the major factor that impacts the performance and reliability of the system is the number of nodes, denoted by k . Theoretically, when k increases the application performance increases, but the reliability decreases. Thus, the performance improvement can be determined as

$$P_r = \frac{T_S - T_P}{T_S} = 1 - \frac{T_P}{T_S}, \quad (5.1)$$

where T_S is the completion time on a single-node, and T_P is the completion time on a k -node system. For single kernel GPU programming, T_S is the sum of the kernel execution time (T_K), host-to-device memory copy (T_{HD}), device-to-host memory copy (T_{DH}), the execution time on the host (T_C), and the overhead caused by parallelization, such as data communication between nodes (T_{PO}) [36].

For a node-wise GPU environment, the sequential completion time is defined as

$$T_S = T_K + T_{HD} + T_{DH} + T_C. \quad (5.2)$$

Assuming that the computation is data parallelizable with load balancing (the data to be executed are equally distributed across the compute nodes), T_K is completely parallelizable; i.e., it is divisible by k . Yet, the data transfer time between the CPU and the GPU on each node depends on the amount of data the node receives. Therefore, only parts of T_{HD} and T_{DH} can be reduced by the distribution. Moreover, T_C can also be partially divisible by k . Let f_{HD} , f_{DH} , and f_C be the fractions of T_{HD} , T_{DH} , and T_C that can be distributed, respectively. Amdahl's law [37] suggests that

$$T_P \geq \frac{1}{k} (T_K + f_{HD}T_{HD} + f_{DH}T_{DH} + f_C T_C) + (1 - f_{HD})T_{HD} + (1 - f_{DH})T_{DH} + (1 - f_C)T_C + T_{PO}. \quad (5.3)$$

To describe the system reliability, the time-to-failure (TTF) for each node is assumed to follow a Weibull distribution. For simplicity and without loss of generality, the failure rates are assumed to be equal ($\lambda = \lambda_1 = \lambda_2 = \lambda_3 = \dots = \lambda_k$). Also, once a node fails, the entire computation is assumed to fail. Therefore, the probability that the system will survive beyond time t is expressed as

$$S(t) = e^{-k\lambda t^c}, \quad (5.4)$$

where t is measured from the moment that the system starts until the application is completed, and c is the shape parameter of the Weibull distribution for each node [14], [15].

For the purpose of estimating T_P and $S(t)$, where $t = T_P$, the proportions of T_K , T_{HD} , T_{DH} , T_C , and T_{PO} to T_S are assumed to be 0.75, 0.02, 0.01, 0.20, and 0.02, respectively. However, these numbers are just an example to demonstrate the applicability of the model. The parallelizable fractions are $f_{HD} = 1$, $f_{DH} = 1$, and $f_C = 0.90$. Moreover, T_S is scaled to an extended period of time (e.g., 15 days) in order to study the impact of the system reliability.

To determine if an application has acceptable performance and reliability, the graphs of the application performance against the system reliability are plotted. Figure 5.1 shows the probability of survival for different values of the Weibull shape parameter, c , the number of nodes, k , and the failure rates, λ . The y-axis represents the survival probability of the system. The x-axis represents the number of nodes in a logarithmic scale. The percentage of performance improvement for each value of k is presented in parentheses beneath $\log_{10} k$.

In Figure 5.1, the study shows that the system reliability, $S(t)$, can increase with a growth in k before it ultimately decreases. This can be interpreted as being due to the fact that T_P depends on k . The completion time T_P decreases with an increase in k . Moreover, as c becomes larger, the system tends to be less reliable. There is evidence, [14] and [15], that time-to-failure follows a Weibull distribution. It is known that for large c values, the Weibull becomes more symmetric, approximating a normal distribution. Hence, in this work, we consider the values of c between 1 and 2, where the Weibull failure rate decreases with time ($c > 1$) and c is not large enough for the Weibull to approximate normal. Additionally, we consider the system from the start to the first failure.

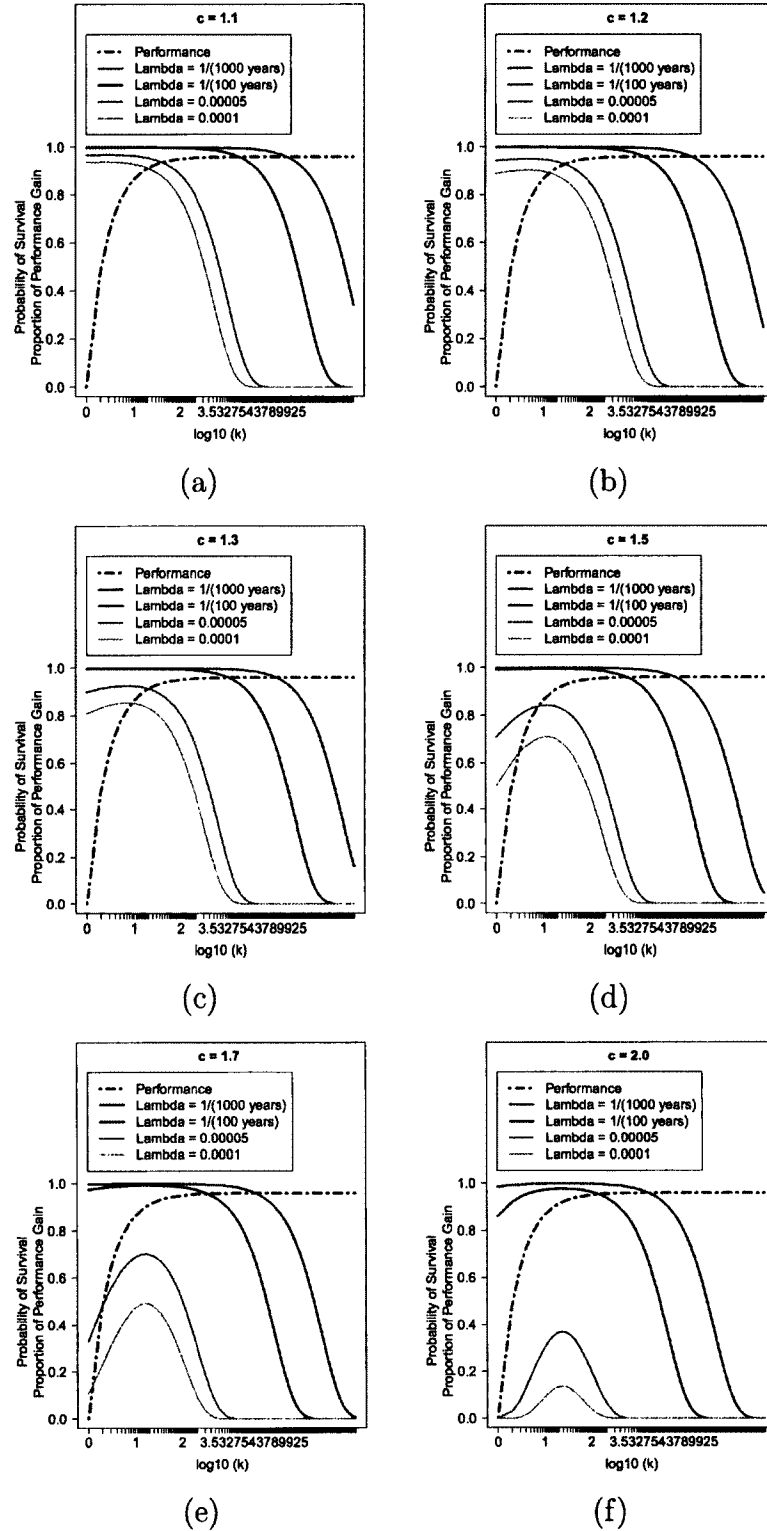


Figure 5.1: Probability of survival for different values of c and k from Equation (5.4)

The graphs in Figure 5.1 also show the system reliability of the given application for different values of λ . Since the system reliability decreases and application performance increases with a growth in the number of nodes, one can identify a number of nodes k that has acceptable levels of reliability and performance. For instance, from Figure 5.1, one can choose, for a given curve, a value for k that would give an acceptable reliability and performance. For example, it is seen from the curve in Figure 5.1 (a), for $\lambda = 1/1000$ yrs that one can choose a k value with performance close to 96 percent that has a reliability close to 1.

5.1.1 Predicting the System Size from the Maximum System Reliability

Let p be the parallelizable part of the program, defined by

$$p = T_K + f_{HD} T_{HD} + f_{DH} T_{DH} + f_C T_C, \quad (5.5)$$

and s be the sequential (non-parallelizable) part of the program, defined by

$$s = (1 - f_{HD}) T_{HD} + (1 - f_{DH}) T_{DH} + (1 - f_C) T_C + T_{PO}. \quad (5.6)$$

Therefore, by Amdahl's law [37], the parallel application completion time can be expressed as

$$T_P = \frac{p}{k} + s. \quad (5.7)$$

To find the optimum k that maximizes $S(T_P, k)$, one must solve the following equation

$$\frac{\partial}{\partial k} S(T_P, k) = -\lambda (k c T_P^{c-1} T'_P + T_P^c) e^{-k\lambda T_P^c} = 0,$$

$$(k c T'_P + T_P) T_P^{c-1} = 0.$$

Because $T'_P = -p/k^2$,

$$k c T'_P + T_P = -\frac{cp}{k} + \left(\frac{p}{k} + s\right) = 0.$$

Hence,

$$k = \frac{(c-1)p}{s}. \quad (5.8)$$

Since k must be at least 1, Equation (5.8) indicates that, for the system reliability with a Weibull-distributed failure intensity, c must be larger than 1. In addition, the proportion of p over s must also be at least $1/(c-1)$. Furthermore, one can predict the improvement in application performance from Equations (5.1) and (5.3).

5.1.2 Predicting the System Size from an Expected Performance

In the previous section, we have proposed a model to determine the optimum k that gives the maximum system reliability. However, typical programmers focus on the performance improvement without considering the system reliability when deploying parallel programming on a large scale system. In this section, a model that identifies the optimal k value for a desired performance level is proposed.

The application problems are categorized into two cases: non-scalable and scalable workload. Since Amdahl's law is used for explaining the speed improvement when the execution time is related to the problem size, Amdahl's law is applied to the former case. For the latter case, Gustafson's law is used because Gustafson's law states that for a scalable problem that maintains a fixed execution time, the speedup is a linear function of system size.

5.1.2.1 Non-scalable Workload

For a fixed problem size, the workload does not grow when more nodes are deployed. Therefore, the completion time decreases as the number of nodes increases. According to Amdahl's law [37], $T_S = p + s - T_{PO}$, and $T_P = p/k + s$. Since $P_r = 1 - T_P/T_S$, $T_P = T_S(1 - P_r)$. As a result,

$$k = \frac{p}{T_S(1 - P_r) - s}, \quad (5.9)$$

where $T_S(1 - P_r) - s > 0$. Hence, $P_r < 1 - s/T_S$.

5.1.2.2 Scalable Workload

For a scalable problem, more nodes are added to the system as the problem size increases. Gustafson's [38], [39] suggests that if the workload is scalable according to the number of nodes, T_P is fixed, while T_S is scaled by k . Let $T_P = T_K + T_{HD} + T_{DH} + T_C + T_{PO} = p + s$. Consequently,

$$\begin{aligned} T_S &= k(T_K + f_{HD}T_{HD} + f_{DH}T_{DH} + f_C T_C) \\ &\quad + (1 - f_{HD})T_{HD} + (1 - f_{DH})T_{DH} + (1 - f_C)T_C \\ &= kp + s + T_{PO}. \end{aligned}$$

From $P_r = 1 - T_P/T_S$, $T_S = T_P/(1 - P_r)$. Thus,

$$k = \frac{1}{p} \left(\frac{p + s}{1 - P_r} - s + T_{PO} \right), \quad (5.10)$$

where $P_r \neq 1$.

Once k and T_P are determined by the models indicated by Equations (5.9), and (5.10), the system reliability can be predicted by Equation (5.4). However, if

the reliability is unacceptably low, meaning that the survival probability is below the baseline specified by the programmer, the checkpoint/restart mechanism may be needed in order to increase the system reliability.

5.2 Checkpoint Scheduling

In a GPGPU environment, a checkpoint will have to be performed on both GPU and CPU during the GPGPU kernel execution. Besides, a typical checkpoint is performed only on the CPU. In addition, to maintain data correctness, a checkpoint must not be allowed during data transfer.

To find an optimal checkpoint placement, we enhance an original optimal checkpoint model [12], [13] for an HPC system. Let C_G and C_C be checkpoint costs from the GPU to the CPU, and from the CPU to a reliable storage on a k -node system, respectively. However, the GPU checkpoint latency (the time to transfer the data from the GPU to the CPU on a k -node system) can be estimated by the device-to-host memory transfer time. Therefore, C_G is defined as

$$C_G = O_G^C + \frac{f_{DH}T_{DH}}{k} + (1 - f_{DH})T_{DH}, \quad (5.11)$$

where O_G^C is the time spent on a checkpoint process.

Moreover, the proportion of kernel execution on k nodes is T_K/kT_P . The proportion of CPU execution on k nodes is $[f_C T_C + k(1 - f_C)T_C]/kT_P$. Thus, the checkpoint cost for a GPGPU application becomes

$$\begin{aligned} C &= \frac{T_K}{kT_P}(C_C + C_G) + \frac{f_C T_C + k(1 - f_C)T_C}{kT_P} C_C \\ &= \frac{1}{kT_P} [T_K C_G + (T_K + f_C T_C + k(1 - f_C)T_C) C_C]. \end{aligned} \quad (5.12)$$

Furthermore, let R_C be the CPU recovery cost of a k -node system. Then, the GPU recovery cost of the k -node system is estimated by

$$R_G = O_G^R + \frac{f_{HD}T_{HD}}{k} + (1 - f_{HD})T_{HD}, \quad (5.13)$$

where O_G^R is the time spent on a recovery process.

Similarly, the recovery cost can be evaluated by

$$R = \frac{1}{kT_P} [T_K R_G + (T_K + f_C T_C + k(1 - f_C) T_C) R_C]. \quad (5.14)$$

Let $n(t)$ be the checkpoint frequency function, such that

$$1 = \int_{t_{m-1}}^{t_m} n(t) dt, \quad (5.15)$$

where t_m , ($m = 1, 2, 3, \dots$) is the m^{th} checkpoint placement, and $t_0 = 0$. Let T_f be a random variable representing the TTF, and φ be a rollback coefficient that is ranged between 0 and 1. Note that the rollback coefficient can be estimated by the algorithm presented by Liu *et al* [12]. The number of checkpoints from the system start until a failure occurs is given by $C \int_0^{T_f} n(t) dt$. Also, according to [12] and [13], the recomputing time can be approximated by

$$T_b \approx \frac{\varphi}{n(T_f)}. \quad (5.16)$$

Consequently, the wasted time is expressed as

$$W(T_f) = C \int_0^{T_f} n(\tau) d\tau + \frac{\varphi}{n(T_f)} + R. \quad (5.17)$$

Let $f(t)$ be the probability density function (pdf) of the system TTF. The expected wasted time becomes

$$E[W] = \int_0^\infty \left[C \int_0^t n(\tau) d\tau + \frac{\varphi}{n(t)} + R \right] f(t) dt. \quad (5.18)$$

Therefore, the optimal checkpoint frequency function that minimizes the expected wasted time can be expressed as

$$n(t) = \sqrt{\frac{\varphi}{C}} \sqrt{\frac{f(t)}{S(t)}}. \quad (5.19)$$

From the probability of survival described by Equation (5.4), the distribution of time-to-failure can be expressed as

$$f(t) = k \lambda c t^{c-1} e^{-k \lambda t^c}. \quad (5.20)$$

Therefore, from Equations (5.4), (5.19), and (5.20), we have

$$n(t) = \sqrt{\frac{\varphi}{C}} \sqrt{k \lambda c t^{c-1}}. \quad (5.21)$$

From Equation (5.15), it is seen that

$$\begin{aligned} 1 &= \int_{t_{m-1}}^{t_m} \sqrt{\frac{\varphi}{C}} \sqrt{k \lambda c t^{c-1}} dt \\ &= \frac{2}{c+1} \sqrt{\frac{k \lambda c \varphi}{C}} \left(t^{\frac{c+1}{2}} \Big|_{t_{m-1}}^{t_m} \right) \\ &= \frac{2}{c+1} \sqrt{\frac{k \lambda c \varphi}{C}} \left(t_m^{\frac{c+1}{2}} - t_{m-1}^{\frac{c+1}{2}} \right). \end{aligned}$$

Thus, t_m can be obtained by

$$t_m = \left(\frac{c+1}{2} \sqrt{\frac{C}{k \lambda c \varphi}} + t_{m-1}^{\frac{c+1}{2}} \right)^{\frac{2}{c+1}}. \quad (5.22)$$

Since $t_0 = 0$,

$$t_1 = \left(\frac{c+1}{2} \sqrt{\frac{C}{k \lambda c \varphi}} \right)^{\frac{2}{c+1}},$$

and

$$t_2 = \left(\frac{c+1}{2} \sqrt{\frac{C}{k \lambda c \varphi}} + \frac{c+1}{2} \sqrt{\frac{C}{k \lambda c \varphi}} \right)^{\frac{2}{c+1}}$$

$$= \left(2 \frac{c+1}{2} \sqrt{\frac{C}{k \lambda c \varphi}} \right)^{\frac{2}{c+1}}.$$

Therefore, by induction, we obtain

$$t_m = \left(m \frac{c+1}{2} \sqrt{\frac{C}{k \lambda c \varphi}} \right)^{\frac{2}{c+1}}. \quad (5.23)$$

That is, for the selected λ and c , the sequence of optimal checkpoint placements t_m , $m = 1, 2, 3, \dots$, where $t_0 = 0$, can be obtained by Equation (5.23).

The graphs that illustrate the influence of k , as predicted by Equations (5.8), (5.9), and (5.10), on the checkpoint interval, are shown in Figure 5.2.

From Figure 5.2 (a)–(c), the results show that, as k increases, the checkpoint frequency decreases. On the other hand, when the application performance becomes a criterion (Figure 5.2 (d)–(i)), more nodes are needed in order to improve the performance, resulting in lower reliability. Hence, more checkpoints are needed to mitigate the effects of failures, leading to a smaller checkpoint interval. It can also be seen that the checkpoint interval decreases when λ increases (an increase in λ decreases the reliability of the system). Furthermore, when comparing the graphs horizontally, it is seen that as c increases, the checkpoint interval decreases because the system becomes less reliable. Therefore, more checkpoints are needed.

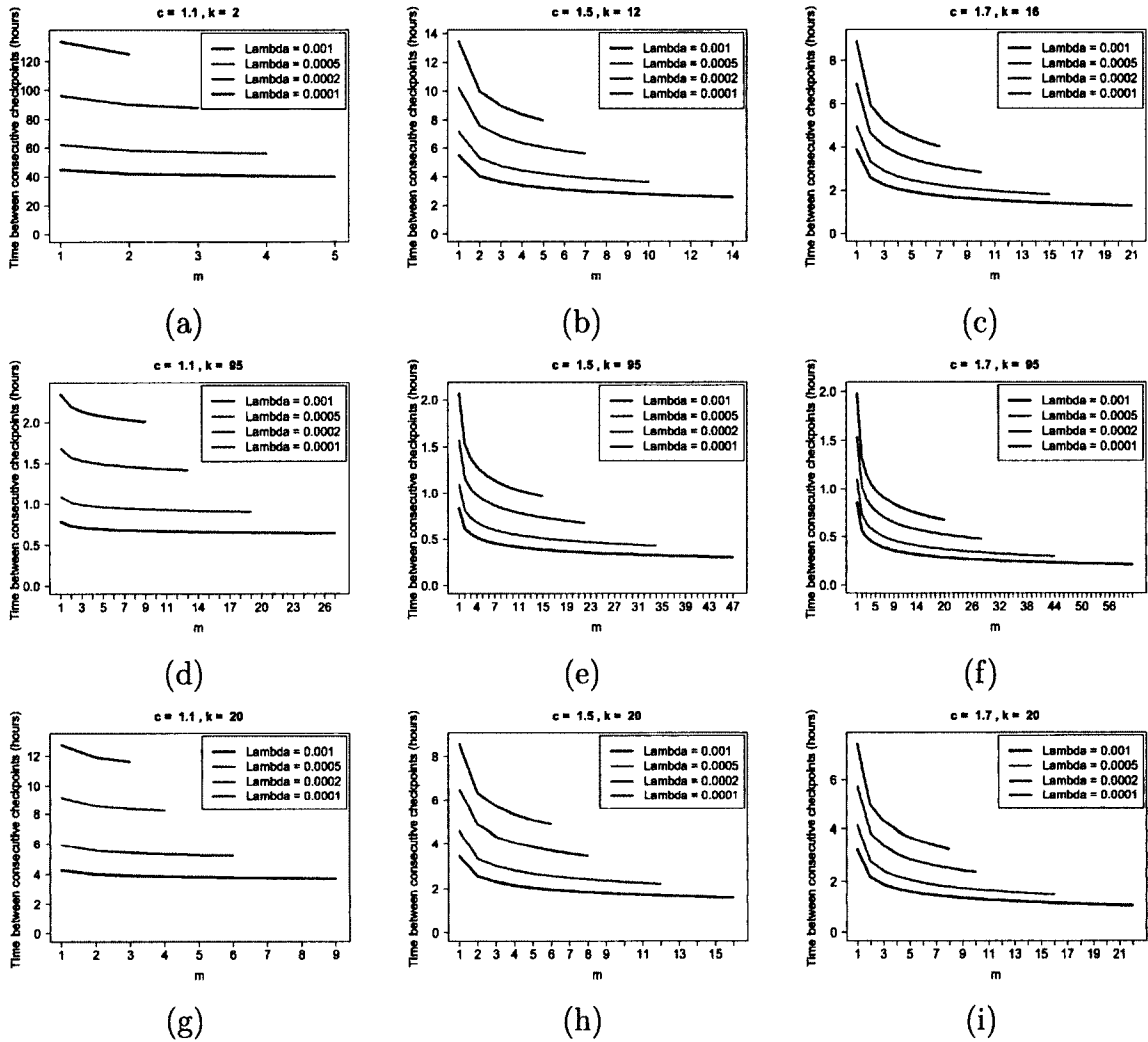


Figure 5.2: The time between two consecutive checkpoints when: (a)–(c) k is predicted by the maximal reliability expressed by Equation (5.8); (d)–(f) k is predicted by the desired performance with fixed workload expressed by Equation (5.9); (g)–(i) k is predicted by the desired performance with scalable workload expressed by Equation (5.10).

5.3 Performance

In Section 5.1, the mathematical models (Equations (5.8), (5.9), and (5.10)) to derive the optimal number of nodes based on different criteria have been presented. The previous models did not consider the effect of a failure on the application performance. However, it is possible that there will be failure occurrence during the

computation. This will impact the application performance due to the recomputing time. As a result, both the effect of reliability and application performance are considered in this section. Thus, the reliability-aware performance model can be denoted as follows:

$$\widehat{P}_r = 1 - \frac{T_P + E[T_{bk}]}{T_S + E[T_{b1}]}, \quad (5.24)$$

where $E[T_{bk}]$ is the expected recomputing time for an application running on k nodes, and T_{b1} is the expected recomputing time for an application running on a single node.

These recomputing times can be expressed as

$$E[T_{bk}] = \frac{\int_0^{T_P} t f(t, k) dt}{\int_0^{T_P} f(t, k) dt} \quad (5.25)$$

$$= \frac{\frac{1}{c(k\lambda)^{1/c}} \gamma\left(\frac{1}{c}, k\lambda T_P^c\right) - \frac{1}{k\lambda} T_P^{1-c} e^{-k\lambda T_P^c}}{1 - e^{-k\lambda T_P^c}}, \quad (5.26)$$

and

$$E[T_{b1}] = \frac{\frac{1}{c\lambda^{1/c}} \gamma\left(\frac{1}{c}, \lambda T_P^c\right) - \frac{1}{\lambda} T_P^{1-c} e^{-\lambda T_P^c}}{1 - e^{-\lambda T_P^c}}, \quad (5.27)$$

where $\gamma(z, y)$ is a lower incomplete gamma function, $\gamma(z, y) = \int_0^y t^{z-1} e^{-t} dt$.

Despite the costs of checkpoint and recovery, it would be of interest to determine if the checkpoint/restart mechanism can increase the application performance by shortening the recomputing time.

Let $E[W_k]$ and $E[W_1]$ be the expected wasted time on a k -node and a single-node systems, respectively. If the i^{th} node is replaced at the j^{th} restart, by substituting

Equations (5.20) and (5.21) into Equation (5.18), we obtain

$$\begin{aligned}
E[W_k] &= \int_0^\infty \left[C \int_0^t \sqrt{\frac{\varphi}{C}} \sqrt{k \lambda c \tau^{c-1}} d\tau + \varphi \sqrt{\frac{C}{\varphi k \lambda c t^{c-1}}} + R \right] k \lambda c t^{c-1} e^{-k \lambda t^c} dt \\
&= \int_0^\infty \frac{2}{c+1} \sqrt{C \varphi k \lambda c} t^{\frac{c+1}{2}} k \lambda c t^{c-1} e^{-k \lambda t^c} dt \\
&\quad + \int_0^\infty \sqrt{\frac{C \varphi}{k \lambda c t^{c-1}}} k \lambda c t^{c-1} e^{-k \lambda t^c} dt \\
&\quad + \int_0^\infty R k \lambda c t^{c-1} e^{-k \lambda t^c} dt. \tag{5.28}
\end{aligned}$$

Let $u = k \lambda t^c$. Thus,

$$\begin{aligned}
E[W_k] &= \frac{2}{c+1} \sqrt{C \varphi k \lambda c} \int_0^\infty \left(\frac{u}{k \lambda}\right)^{\frac{c+1}{2c}} e^{-u} du \\
&\quad + \sqrt{\frac{C \varphi}{k \lambda c}} \int_0^\infty \left(\frac{u}{k \lambda}\right)^{-\frac{c-1}{2c}} e^{-u} du \\
&\quad + R \int_0^\infty e^{-u} du \\
&= \frac{2}{c+1} \cdot \frac{\sqrt{C \varphi k \lambda c}}{(k \lambda)^{\frac{c+1}{2c}}} \int_0^\infty u^{\left(\frac{c+1}{2c}+1\right)-1} e^{-u} du \\
&\quad + \sqrt{\frac{C \varphi}{k \lambda c}} (k \lambda)^{\frac{c-1}{2c}} \int_0^\infty u^{\frac{c+1}{2c}-1} e^{-u} du \\
&\quad - R e^{-u} \Big|_0^\infty. \tag{5.29}
\end{aligned}$$

Therefore,

$$E[W_k] = \frac{2}{c+1} \cdot \frac{\sqrt{C \varphi c}}{(k \lambda)^{\frac{1}{2c}}} \Gamma\left(\frac{c+1}{2c} + 1\right) + \sqrt{\frac{C \varphi}{c}} \cdot \frac{1}{(k \lambda)^{\frac{1}{2c}}} \Gamma\left(\frac{c+1}{2c}\right) + R, \tag{5.30}$$

where $\Gamma(z)$ is a gamma function.

From the property of the gamma function ($\Gamma(z+1) = z \Gamma(z)$), we have

$$\Gamma\left(\frac{c+1}{2c} + 1\right) = \frac{c+1}{2c} \Gamma\left(\frac{c+1}{2c}\right). \text{ Hence,}$$

$$E[W_k] = \frac{2}{(k \lambda)^{\frac{1}{2c}}} \sqrt{\frac{C \varphi}{c}} \Gamma\left(\frac{c+1}{2c}\right) + R. \tag{5.31}$$

Similarly,

$$E[W_1] = \frac{2}{\lambda^{\frac{1}{2c}}} \sqrt{\frac{C\varphi}{c}} \Gamma\left(\frac{c+1}{2c}\right) + R. \quad (5.32)$$

The application performance can be expressed as

$$\bar{P}_r = 1 - \frac{T_P + E[W_k]}{T_S + E[W_1]}. \quad (5.33)$$

Therefore, the change in performance is given by the following expression:

$$\bar{P}_r - \hat{P}_r = \frac{T_P + E[T_{bk}]}{T_S + E[T_{b1}]} - \frac{T_P + E[W_k]}{T_S + E[W_1]}. \quad (5.34)$$

To study the impacts of checkpoints on the application performance for the predicted k values, Figure 5.3 illustrates the reliability-aware performance that the application will achieve during deploying or not deploying checkpoints (evaluated by Equations (5.24) and (5.33)). Again, Figure 5.3 (a)–(c) illustrates the performance for various values of c and λ when k is predicted by Equation (5.8); Figure 5.3 (d)–(f) shows the performance when k is predicted by Equation (5.9); and Figure 5.3 (g)–(i) presents the performance when k is predicted by Equation (5.10). It is obvious that, on a large scale system, when deploying checkpoints, the application performance is better than it will be without checkpoints.

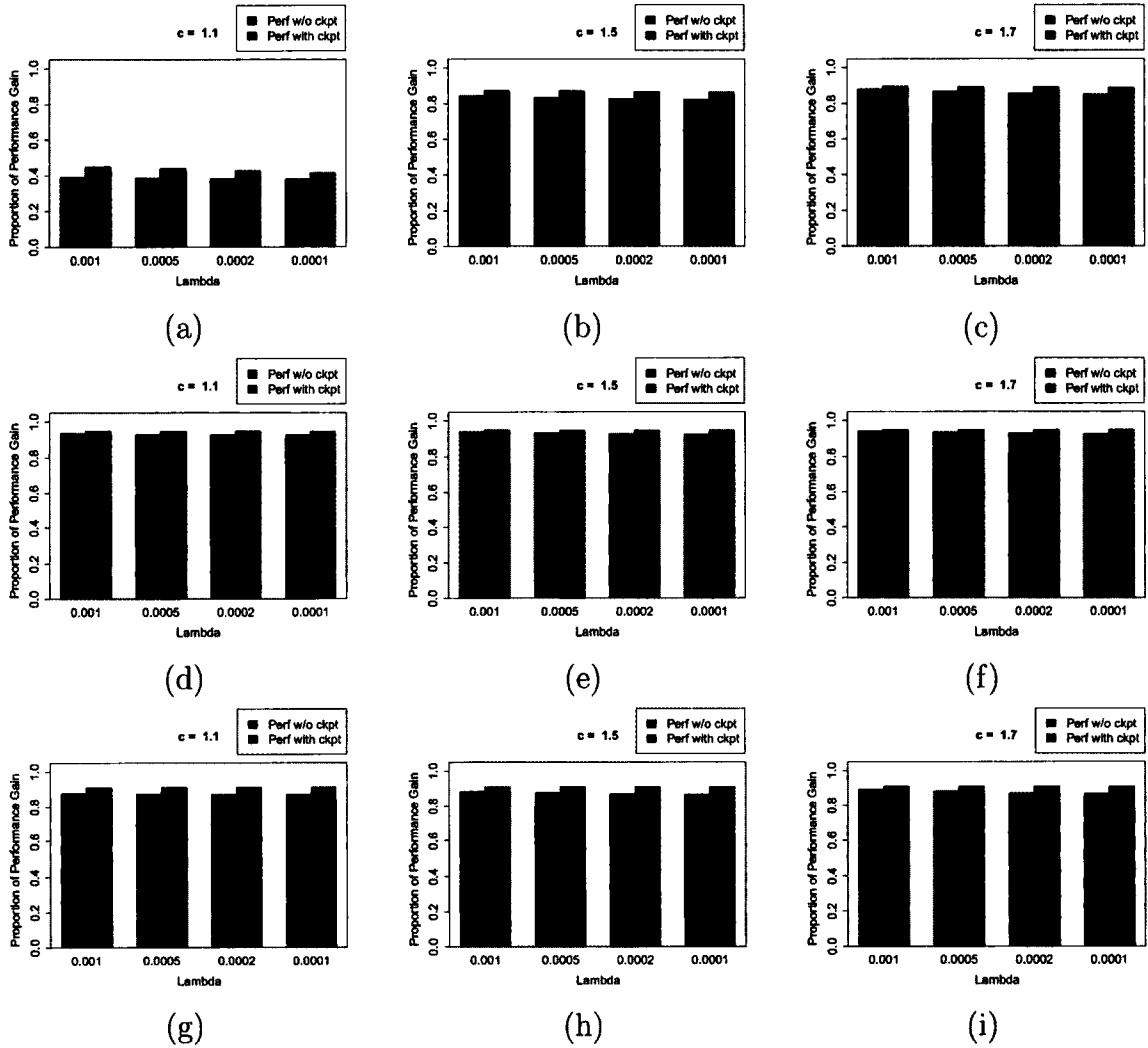


Figure 5.3: Comparison of application performance when: (a)–(c) k is predicted by the maximal reliability expressed by Equation (5.8); (d)–(f) k is predicted by the desired performance with fixed workload expressed by Equation (5.9); (g)–(i) k is predicted by the desired performance with scalable workload expressed by Equation (5.10).

Additionally, in Figure 5.3 (a), the system can gain less performance than that shown in other figures because, for the maximal reliability, Equation (5.8) suggests very few number of nodes.

5.4 Real-World Case Study

In this simulation, the optimized matrix multiplication algorithm given by [30] is analyzed as a case study, where the matrix size is $2^{23} \times 2^{23}$ elements. Therefore, T_S is approximately 30 days; i.e., twice as long as the previous example. Moreover, the proportions of T_K , T_{HD} , T_{DH} , T_C , and T_{PO} to T_S are 0.7458, 0.0122, 0.0172, 0.2247, and 0.05, respectively. In addition, $f_{HD} = f_{DH} = 1$ and $f_C = 0.78$. These numbers are obtained by the performance model and code analysis presented in [40]. Furthermore, we assume that both host and device memory units are very large.

To illustrate the influence of k values that are predicted by Equations (5.8), (5.9), and (5.10) on the checkpoint interval, we plot the graphs shown in Figure 5.4.

Similar to Figure 5.2, from Figure 5.4 (a)–(c), our study shows that, as k increases with c , the checkpoint interval decreases. Although, as shown by Figure 5.4 (d) – (i), k does not increase with c , more nodes are suggested in order to improve the performance. As a result, the reliability decreases, and more checkpoints are needed, which leads to smaller checkpoint intervals. Furthermore, as λ is also a critical factor in the reliability model, when λ is very small; for instance, $\lambda = 1/(1000\text{yrs})$, checkpoints may not be needed (as shown by Figure 5.4 (a) and (g)). On the other hand, when λ is large, the checkpoint interval is very small, indicating that the system may take too much computation for checkpointing and may cause a performance drop.

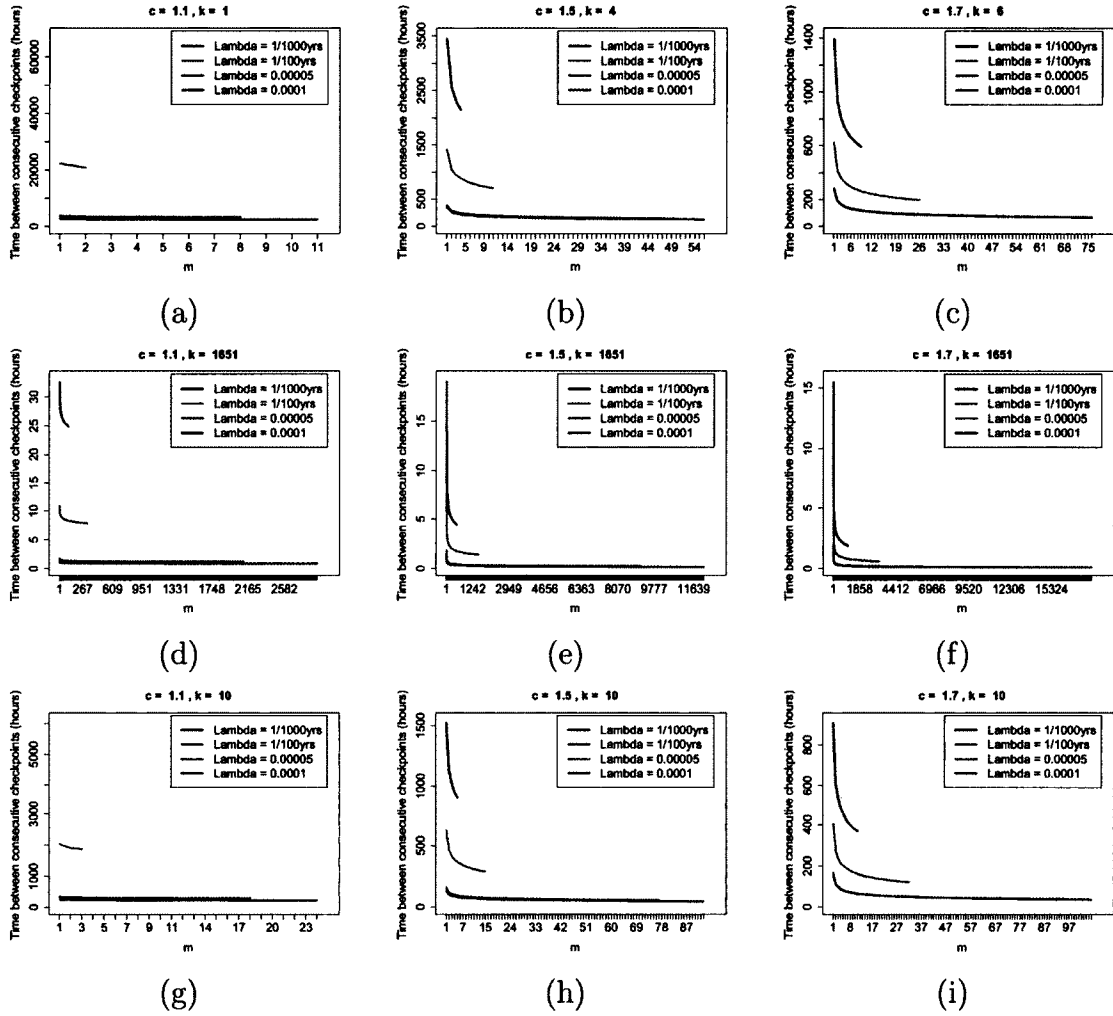


Figure 5.4: Time between two consecutive checkpoints for the case study when: (a)–(c) k is predicted by the maximal reliability expressed by Equation (5.8); (d)–(f) k is predicted by the desired performance with fixed workload expressed by Equation (5.9); (g)–(i) k is predicted by the desired performance with scalable workload expressed by Equation (5.10).

To study the impacts of checkpoints on the application performance for the predicted k values, Figure 5.5 illustrates the reliability-aware performance that the application will achieve during deploying or not deploying checkpoints (evaluated by Equations (5.24) and (5.33)).

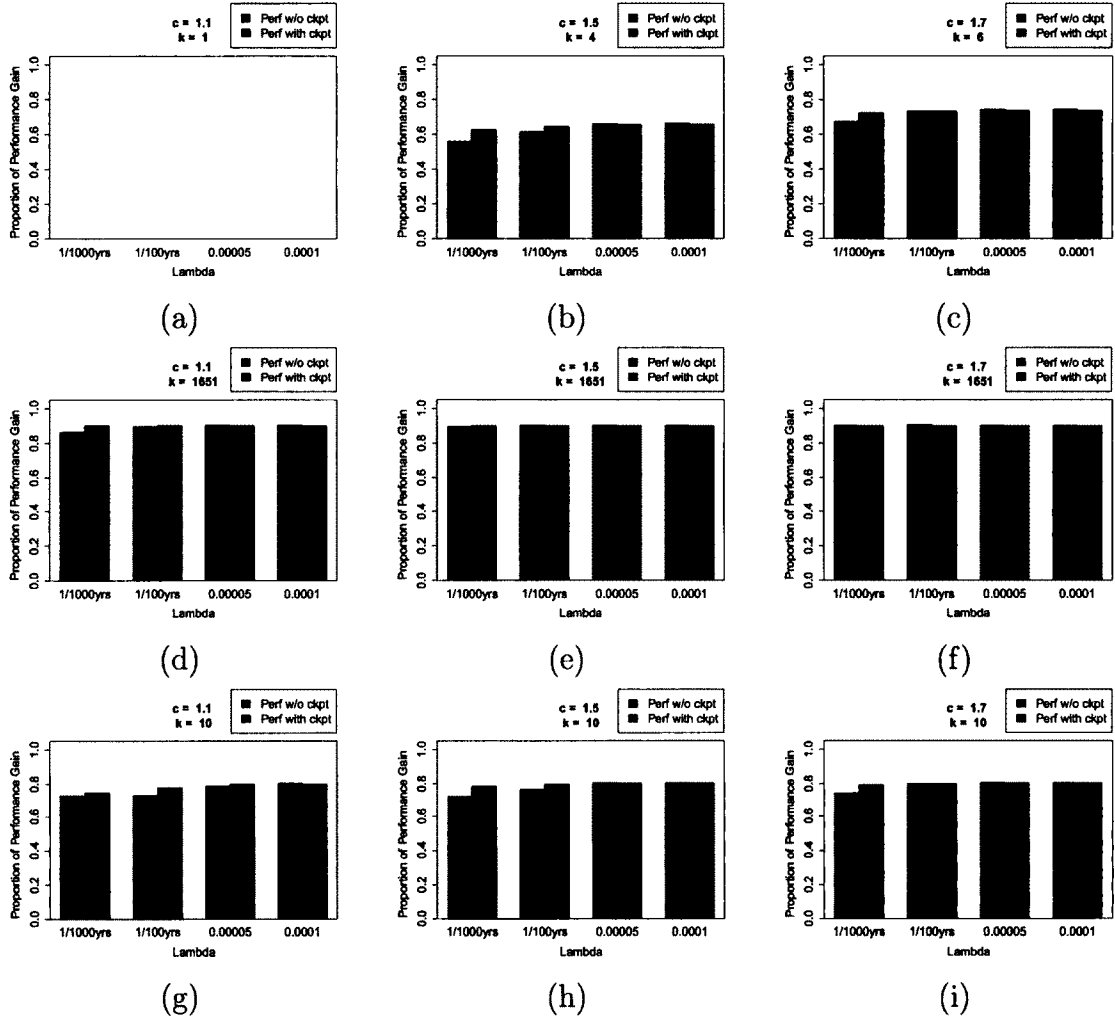


Figure 5.5: Comparison of application performance for the case study when: (a)–(c) k is predicted by the maximal reliability expressed by Equation (5.8); (d)–(f) k is predicted by the desired performance with fixed workload expressed by Equation (5.9); (g)–(i) k is predicted by the desired performance with scalable workload expressed by Equation (5.10).

Figure 5.5 (a)–(c) illustrates the performance for various values of c and λ , when k is predicted by Equation (5.8). As Equation (5.8) gives $k = 1$ when $c = 1.1$, there is no performance gain. Besides, since k increases with c , the performance in Figure 5.5 (c) is larger than in Figure 5.5 (b). Nonetheless, the checkpoint can be advantageous only when λ and c are small.

Figure 5.5 (d)–(f) shows the performance when k is predicted by Equation (5.9). In this case, k is large, leading to high performance but very low reliability. The system spends too much time handling the checkpoint processes. Therefore, the checkpoint can be beneficial only when c is small enough and λ is very small.

Similarly, Figure 5.5 (g)–(i) presents the performance, when k is predicted by Equation (5.10). Since k suggested by Equation (5.10) is not as large as that by Equation (5.9), the performance is lower. However, in this case, the checkpoint can be more advantageous when λ is small.

5.5 The Model with an Excess Weibull

In Section 5.1, the system that can function until the first failure has been considered. In reality, however, after a failure occurs at one node, the other nodes continue to function. As a result, after the j^{th} restart, the time-to-failure, x_{ij} , of the i^{th} node that did not fail has an excess life distribution [14], [15]. Hence, the failure intensity can be described as

$$f(t) = \begin{cases} k \lambda c t^{c-1} e^{-k\lambda t^c} & \text{if the } i^{th} \text{ node is replaced} \\ & \text{at the } j^{th} \text{ restart,} \\ \lambda c \sum_{i=1}^k (x_{ij} + t)^{c-1} e^{-\lambda \sum_{i=1}^k [(x_{ij} + t)^c - x_{ij}^c]} & \text{if the } i^{th} \text{ node is not replaced} \\ & \text{at the } j^{th} \text{ restart,} \end{cases} \quad (5.35)$$

and the probability of survival can be expressed as

$$S(t) = \begin{cases} e^{-k\lambda t^c} & \text{if the } i^{th} \text{ node is replaced at the } j^{th} \text{ restart,} \\ e^{-\lambda \sum_{i=1}^k [(x_{ij} + t)^c - x_{ij}^c]} & \text{if the } i^{th} \text{ node is not replaced at the } j^{th} \text{ restart.} \end{cases} \quad (5.36)$$

If the i^{th} node is not replaced at the j^{th} restart, the checkpoint frequency function becomes

$$n(t) = \sqrt{\frac{\varphi}{C}} \sqrt{\lambda c \sum_{i=1}^k (x_{ij} + t)^{c-1}}. \quad (5.37)$$

Again, substituting Equation (5.37) into Equation (5.15), we obtain

$$1 = \sqrt{\frac{\varphi \lambda c}{C}} \int_{t_{m-1}}^{t_m} \sqrt{\sum_{i=1}^k (x_{ij} + t)^{c-1}} dt. \quad (5.38)$$

Equation (5.38) can be solved numerically by varying t_m and t_{m-1} . Moreover, the performance of the application can be evaluated as described by Equations (5.24) and (5.33). However, in this case, the expected recomputing time on a k -node system, $E[T_{bk}]$, becomes

$$E[T_{bk}] = \frac{\lambda c \int_0^{T_P} t \sum_{i=1}^k (x_{ij} + t)^{c-1} e^{-\lambda \sum_{i=1}^k [(x_{ij} + t)^c - x_{ij}^c]} dt}{1 - e^{-\lambda \sum_{i=1}^k [(x_{ij} + T_P)^c - x_{ij}^c]}}. \quad (5.39)$$

For a single-node system, the excess time after the j^{th} restart is denoted by x_j . Therefore, $E[T_{b1}]$ becomes

$$E[T_{b1}] = \frac{\frac{e^{\lambda x_j^c}}{c \lambda^{\frac{1}{c}}} \left[\gamma\left(\frac{1}{c}, \lambda(x_j + T_P)^c\right) - \gamma\left(\frac{1}{c}, \lambda x_j^c\right) \right] - T_P e^{-\lambda[(x_j + T_P)^c - x_j^c]}}{1 - e^{-\lambda[(x_j + T_P)^c - x_j^c]}}, \quad (5.40)$$

where $\gamma(z, y)$ is a lower incomplete gamma function.

In addition, The expected wasted time $E[W_k]$ and $E[W_1]$ becomes

$$\begin{aligned} E[W_k] &= \sqrt{C \varphi \lambda^3 c^3} e^{\lambda \sum_{i=1}^k x_{ij}^c} \\ &\int_0^\infty \left[\sum_{i=1}^k (x_{ij} + t)^{c-1} e^{-\lambda \sum_{i=1}^k (x_{ij} + t)^c} \int_0^t \sqrt{\sum_{i=1}^k (x_{ij} + \tau)^{c-1}} d\tau \right] dt \\ &+ \sqrt{C \varphi \lambda c} e^{\lambda \sum_{i=1}^k x_{ij}^c} \int_0^\infty \sqrt{\sum_{i=1}^k (x_{ij} + t)^{c-1}} e^{-\lambda \sum_{i=1}^k (x_{ij} + t)^c} dt \\ &+ R, \end{aligned} \quad (5.41)$$

and

$$E[W_1] = \frac{2}{\lambda^{\frac{1}{2c}}} \sqrt{\frac{C\varphi}{c}} e^{\lambda x_j^c} \left[\Gamma\left(\frac{c+1}{2c}\right) - \gamma\left(\frac{c+1}{2c}, \lambda x_j^c\right) \right] + R. \quad (5.42)$$

By solving Equations (5.39), (5.40), (5.41), and (5.42) numerically, one can determine, from Equation (5.34), the change in application performance due to the checkpoint/restart mechanism.

5.6 Conclusion

In the real world, application performance and system reliability are key factors that influence large scale HPC applications. However, as the application performance increases with the number of nodes, the system reliability decreases. The impacts of these two critical factors on a heterogeneous HPC system, where the failure intensity of each node follows a Weibull distribution, are studied. Several models have been proposed in order to determine an optimal number of nodes based on three different criteria: maximal reliability, the desired performance of an application with a fixed problem size, and the desired performance of an application with a scalable problem size. In addition, a checkpoint scheduling model for a heterogeneous system with k nodes has been presented. Moreover, we have shown that the checkpoint/restart mechanism based on our checkpoint scheduling model can increase the performance of an application when failures occur as long as c and λ are small.

CHAPTER 6

GPGPU OPTIMIZATION

One of the major concerns in GPGPU optimization is coalescing in global memory. The data to be operated by the GPU multiprocessors are stored in the GPU's DRAM called global memory. Since the global memory is an off-chip memory unit, global memory access is very expensive. In some cases [3], it may take over 400 clock cycles to read coalesced data (the data that are aligned properly in the global memory) required by all threads in a “warp”, which is a group of threads that execute instructions simultaneously (See more detail in Section 6.1.). For uncoalesced data access, it will take much longer due to multiple round-trip accesses. Consequently, uncoalesced global memory access prolongs the memory latency, resulting in a decrease in GPGPU application performance [3].

There are several existing techniques that attempt to reduce the latency due to uncoalesced global memory access. A common way to solve this issue is to promote the use of shared memory. However, shared memory is much smaller than global memory and can lead to bank conflicts in shared memory. This happens when the bandwidth of the shared memory is too small to serve multiple data accesses at a time [3].

Another performance improvement technique is memory rearrangement. This technique aims to re-align data in global memory such that they are coalesced before data access during kernel execution.

This research proposes memory rearrangement techniques to remedy uncoalesced global memory access patterns by using 2-dimensional matrix transpose and 3-dimensional matrix permutation. The proposed techniques can be applied to many common memory access patterns. The cost benefit of these techniques will also be discussed in detail. Moreover, the analytical results reveals that the proposed techniques are beneficial if the transformed array/matrix is frequently accessed.

6.1 Coalescing VS Uncoalescing

To understand the memory access patterns in the GPU, there are two terminologies that have to be introduced in this section: coalescing and uncoalescing.

For the most efficient data access, the data requested by threads in a warp should be aligned properly in the same segment, which is called coalesced memory alignment.

Figure 6.1 illustrates the coalesced and uncoalesced memory access patterns. Figure 6.1 (a) shows the array/matrix elements required by threads in a warp (represented by light blue arrows) are aligned in the same segment 0 – 31. This access pattern is called coalesced memory access. On the other hand, Figure 6.1 (b) shows a single stride uncoalesced memory access pattern with an offset of f and a stride of d . That is the array/matrix elements accessed by threads in a warp start at the

f^{th} element, and each access is skipped by d elements. Listing 6.1 shows the sample GPU code that generates this single stride memory access pattern.

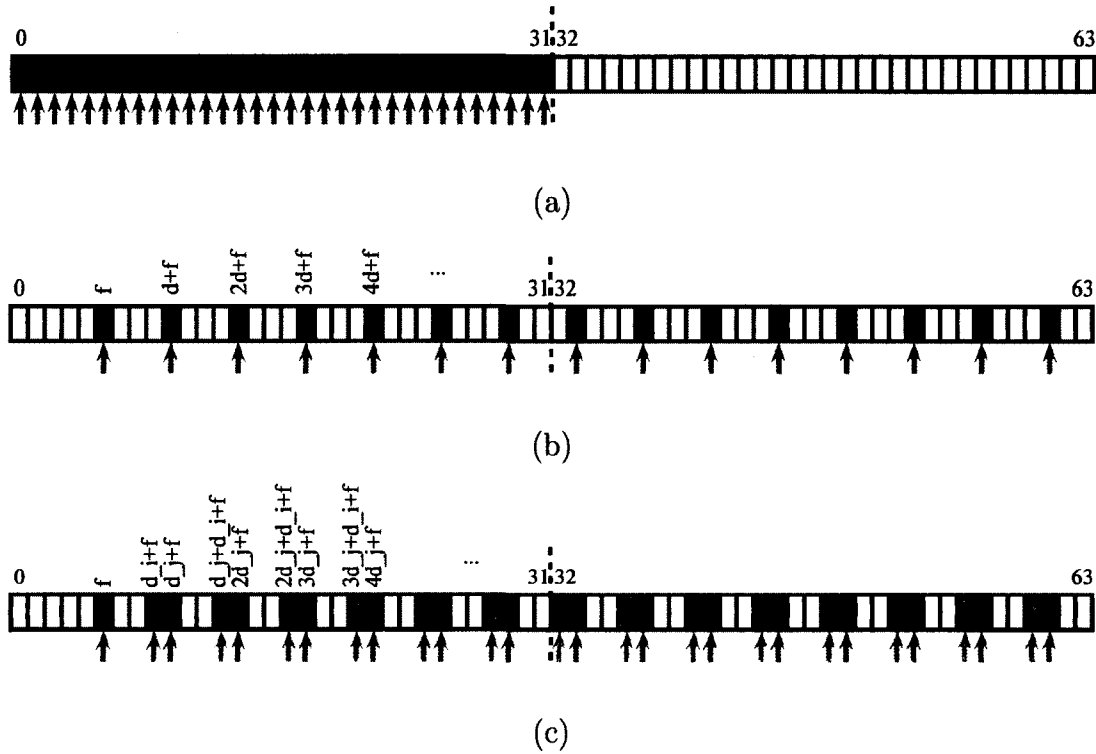


Figure 6.1: Memory access pattern categories: (a) Coalesced memory access, (b) Single stride uncoalesced memory access, and (c) Double stride uncoalesced memory access

Listing 6.1: GPU code that generates a single stride memory access pattern

```

1 idx = blockIdx.x * blockDim.x + threadIdx.x;
2 C[idx] = A[idx] + B[d * idx + f];

```

For a double stride uncoalesced memory access pattern, as illustrated by Figure 6.1 (c), there are two stride parameters, d_i and d_j . Again, the array/matrix elements requested by threads start at the f^{th} element, and the two consecutive accesses are separated by d_j elements. However, the next thread block will request for the set of elements that is d_i separated from the previous set of elements. The sample GPU

code that generates a double stride memory access pattern is shown in Listing 6.3, which is translated from the CPU code with double loops (two iterative parameters) in Listing 6.2.

Listing 6.2: CPU code with a double stride memory access pattern

```

1 for i = 0; i < Count_i; ++i do
2   for j = 0; j < Count_j; ++j do
3     C[i+j] = B[d_i*i + d_j*j];
4   end
5 end

```

Listing 6.3: GPU code that translated from the CPU code in Listing 6.2, which generates a double stride memory access pattern

```

1 xidx = blockIdx.x * blockDim.x + threadIdx.x;
2 yidx = blockIdx.y * blockDim.y + threadIdx.y;
3 C[xidx + (n_C * yidx)] = B[(d_1 * xidx) + (d_2 * n_B * yidx) + f];

```

Since the uncoalesced memory access pattern causes the data needed by a warp to be scattered in the memory, the warp needs multiple round trips in order to access all the data required. Consequently, the memory latency is prolonged, which decreases the application performance.

6.2 Memory Rearrangement

In this section, the techniques for solving uncoalesced global memory, called memory rearrangement, are described. These techniques deploy matrix transpose for a 2D matrix and permutation for a 3D matrix. The uncoalesced memory access problems are categorized by the number of strides. In this research, only two problem cases are discussed: single stride and double stride access patterns.

6.2.1 Single Stride

From the GPU code in Listing 6.1, the elements of matrix B are read by the stride d and the offset f . To eliminate the uncoalesced global memory access, a 2D matrix is constructed such that the width of the 2D matrix is equal to the stride d . Note that, for the best utilization, $Count$ should be a multiple of a warp size, S_W . The example is given by Listing 6.5, which is the GPU code translated from the CPU code in Listing 6.4, and illustrated by Figure 6.2.

Listing 6.4: Array access pattern with a single stride (CPU code)

```

1 for  $i = 0; i < Count; ++i$  do
2    $C[i] = A[i] + B[3 * i + 5];$ 
3 end

```

Listing 6.5: Array access pattern with a single stride (GPU code that is a parallel version of the CPU code in Listing 6.4)

```

1  $idx = blockDim.x * blockIdx.x + threadIdx.x;$ 
2  $C[idx] = A[idx] + B[3 * idx + 5];$ 

```

The CPU code in Listing 6.4 can be written in parallel GPU code as shown in Listing 6.5. The elements of matrix B are read by the stride $d = 3$ and the offset $f = 5$ as shown in Figure 6.2 (a). Figure 6.2 (b) shows the 2D matrix with the width equal to the stride $d = 3$. The 2D matrix is transposed, resulting in the transformed array with the coalesced access pattern illustrated in Figure 6.2 (c).

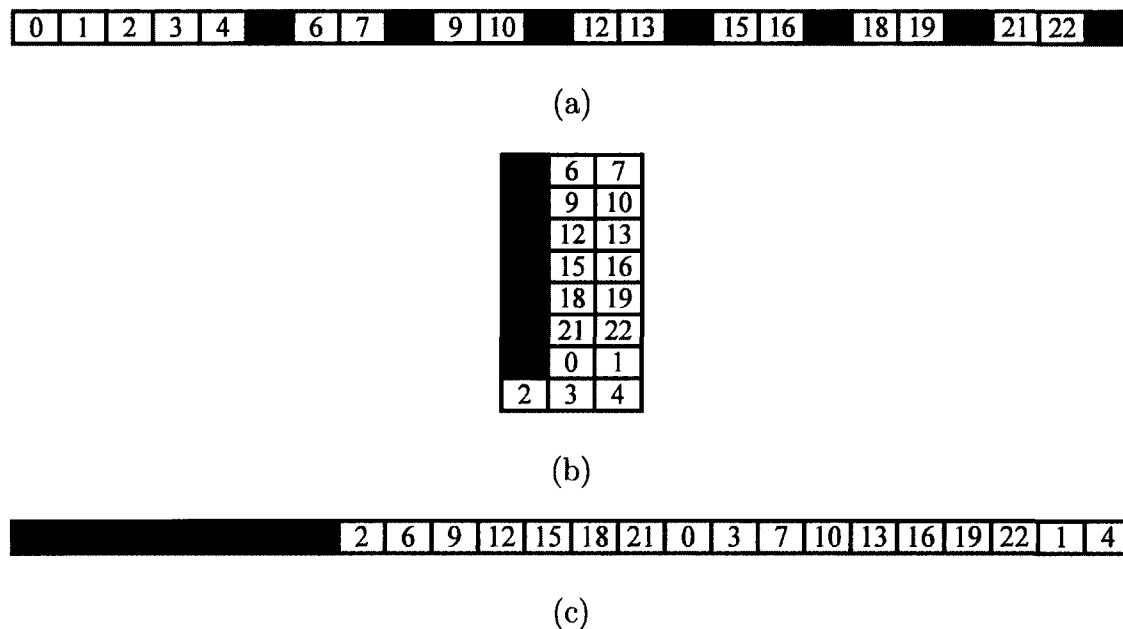


Figure 6.2: Array B transformation associated to the sample code in Listing 6.4 and 6.5 (a) The original array; (b) The 2D matrix is constructed; (c) The transformed array.

Similarly, each of the matrices in Listing 6.6 has a stride $d = 3$. Therefore, the technique illustrated in Figure 6.2 must be applied to arrays A , B , and C .

Listing 6.6: Array access pattern with a single stride in a loop-statement

```

1 for  $i = 5; i < Count; i += 3$  do
2    $C[i] = A[i] + B[i];$ 
3 end

```

In a specific case as shown by Listing 6.7, B is a 2D matrix in which only the elements in the diagonal line will be accessed. Hence, the reference address of an element in B can be determined by $ref = B[in_B + i] = B[(n_B + 1)i]$, where n_B is the width of matrix B . That is, the stride $d = n_B + 1$. Then the technique illustrated in Figure 6.2 can also be applied to this case as shown in Figure 6.3.

Listing 6.7: An access pattern that requires only elements on the diagonal line of a 2D matrix

```

1 for  $i = 0; i < Count; ++i$  do
2    $C[i] = B[i][i];$ 
3 end

```

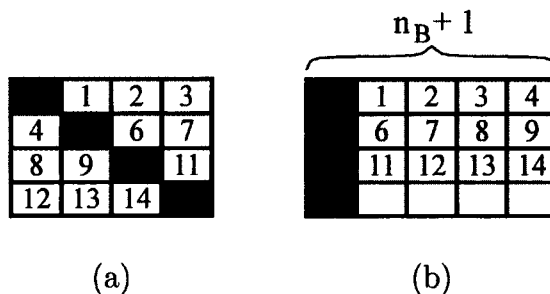


Figure 6.3: Transformation of the 2D matrix that only the elements on the diagonal line are accessed as given by Listing 6.7; (a) Original Layout, (b) Modified matrix with the width of $n_B + 1$.

6.2.2 Double Strides

For this problem category, a 3D matrix structure has to be introduced in order to further describe the double stride memory access patterns.

From a 3D matrix shown in Figure 6.4, the reference address of an element in the matrix can be determined by $ref = xnp + yp + z$, where x is the counter that counts along the height (m) of the matrix, y is the counter that counts along the width (n) of the matrix, and k is the counter that counts along the depth (p) of the matrix. This implies that the threads in a warp will access elements along k -direction. Given a CPU code shown in Listing 6.2 with two iterative parameters, where d_i is the stride on the outer-loop and d_j is the stride on the inner-loop, this problem category can be classified into two sub-categories: when $d_i < d_j$ and $d_i > d_j$.

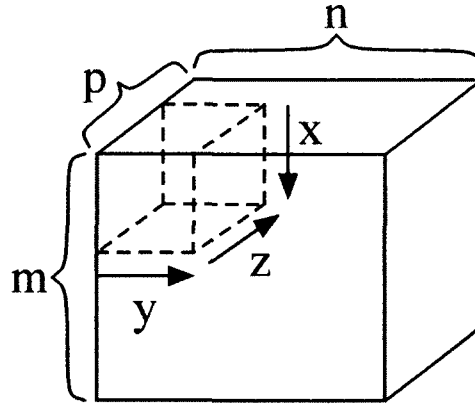


Figure 6.4: 3D Matrix

From the sample code shown in Listing 6.8, there are two strides $d_i = 2$ and $d_j = 6$, i.e., $d_i < d_j$. A 3D matrix should be constructed such that the depth of the matrix (k^{th} -dimension) $p = \lceil \frac{\text{Data.Size}}{d_j} \rceil$, where the height $m = d_i$ and the width $n = d_j/d_i$. In addition, since p implies the number of elements required by threads in a block, p should be equal to a multiple of S_W .

Listing 6.8: An access pattern where $d_i < d_j$

```

1 for i = 0; i < Count_i; ++i do
2   for j = 0; j < Count_j; ++j do
3     C[i+j] = B[2*i + 6*j];
4   end
5 end
```

Figure 6.5 shows the outcome of 3D matrix permutation associated with the sample CPU code in Listing 6.8. From the original layout illustrated by Figure 6.5 (a), the 3D matrix is constructed as shown by Figure 6.5 (b) and permuted by the order of [2,3,1]. Normally, the dimensional order of a 3D matrix is represented by [1,2,3], i.e., 1 represents the height, 2 represents the width, and 3 represents the depth of the 3D matrix, respectively. In this case, the 3D matrix is permuted with the order

of [2,3,1], which means that the height of the matrix becomes the depth, the width becomes the height, and the depth becomes the width. Therefore, the permuted 3D matrix is presented by Figure 6.5 (c), where the final memory layout is shown by Figure 6.5 (d).

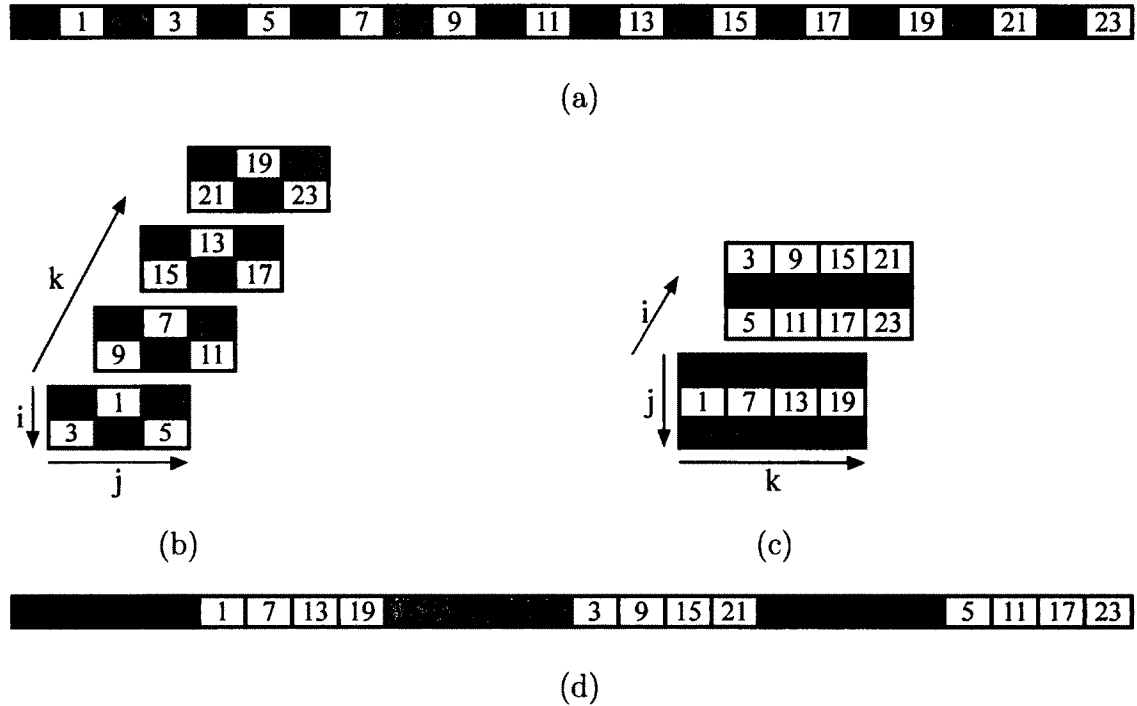


Figure 6.5: 1D array with double strides where $d_i < d_j$; (a) The original layout, (b) The 3D matrix with $m = d_i$, $n = d_j/d_i$, (c) Permuted 3D matrix with the order of [2,3,1], and (d) The final layout.

For the case that $d_i > d_j$ (The sample code is shown by Listing 6.9 and the memory layout is illustrated by Figure 6.6 (a).), the 3D matrix can be constructed such that $m = d_i/d_j$, $n = d_j$, and $p = \left\lceil \frac{Data_Size}{d_i} \right\rceil$ as shown in Figure 6.6 (b). Then the permutation can be performed with the order of [2,1,3] as shown in Figure 6.6 (c). Hence, the outcome of the proposed technique is presented by Figure 6.6 (d).

Listing 6.9: An access pattern where $d_i > d_j$

```

1 for  $i = 0; i < Count\_i; ++i$  do
2   for  $j = 0; j < Count\_j; ++j$  do
3      $C[i+j] = B[8*i + 2*j];$ 
4   end
5 end

```

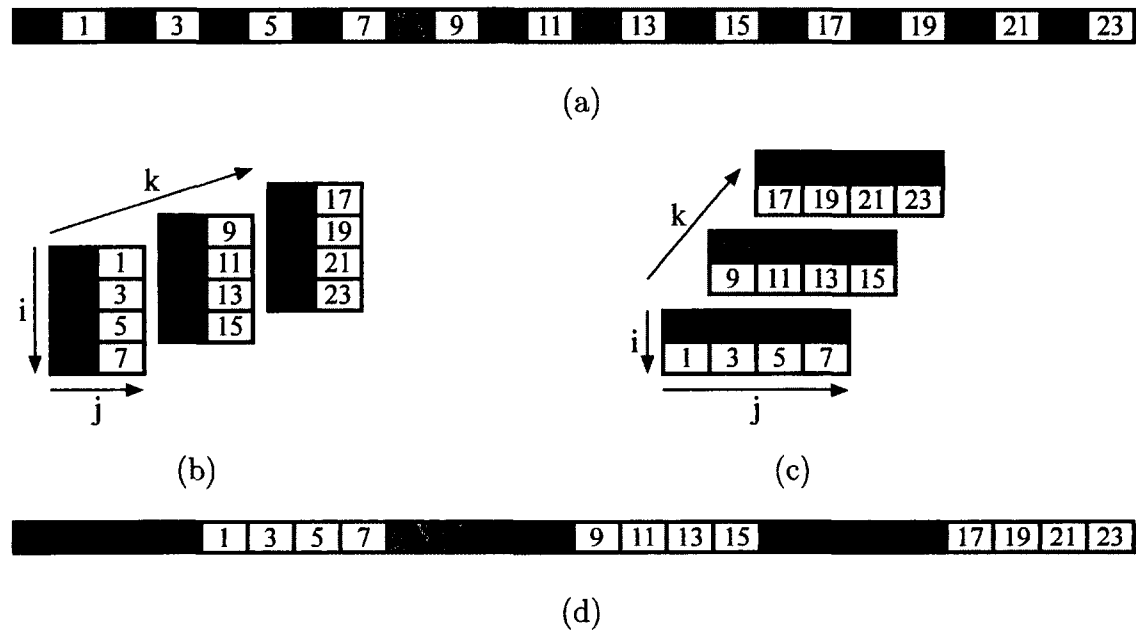


Figure 6.6: 1D array with double strides where $d_i > d_j$; (a) The original layout, (b) The 3D matrix with $m = d_i/d_j$, $n = d_j$, (c) Permuted 3D matrix with the order of $[2,1,3]$, and (d) The final layout.

To describe an access pattern of a 2D matrix as shown by the sample code in Listing 6.10, let the dimension of matrix C be $n_C \times m_C$, and the dimension of matrix B be $n_B \times m_B$. From the code in Listing 6.10, the index of C can be determined by $in_C + j$. If n_C is a multiple of warp size, S_W , C is coalesced. Otherwise, C is uncoalesced and should be reconstructed, such that $n_C \times m_C = n'_C \times m'_C$, where n'_C is equal to a multiple of S_W .

Listing 6.10: Access pattern of a 2D Matrix

```

1 for  $i = 0; i < Count_i; ++i$  do
2   for  $j = 0; j < Count_j; ++j$  do
3      $C[i][j] = B[3*i][2*j];$ 
4   end
5 end

```

As shown by Listing 6.10, the index of B can be determined by $3in_B + 2j$. That is, the strides $d_i = 3n_B$, and $d_j = 2$. Therefore, $d_i > d_j$. Thus, the technique shown in Figure 6.6 can be applied to this case as shown in Figure 6.7. Additionally, every warp will be coalesced if $Count_j$ is the a multiple of S_W .

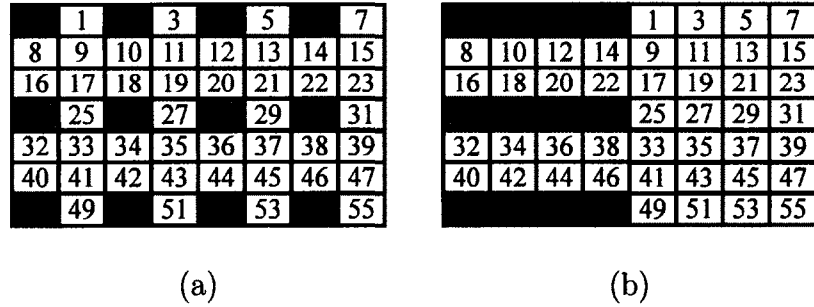


Figure 6.7: 2D matrix with double strides; (a) The original layout, (b) The final layout after 3D matrix permutation with the order of [2,1,3].

After performing the rearrangement process, the outer-loop i can be distributed among thread blocks, while the inner-loop j is distributed among threads in a block.

In summary, the algorithm explaining the proposed technique is given by Listing 6.11. This algorithm describes the approach to solve both single stride and double stride global memory access patterns.

Listing 6.11: An algorithm of matrix transformation to eliminate an uncoalesced global memory access pattern

```

1 Detect whether it is a single-stride or a double-strides access pattern by the
  depths of the nested loop
2 if single-stride then
3   Detect the stride  $d$  and the offset  $f$ 
4   Construct a 2D matrix, where:
5     1.  $n = d$ 
6     2. the first position of the matrix =  $f$ 
7   Transpose the 2D matrix
8 end
9 else if double-strides then
10  Detect the stride  $d_i$  and  $d_j$ , and the the offset  $f$ 
11  if  $d_i < d_j$  then
12    Construct a 3D matrix, where:
13      1.  $m = d_i$ 
14      2.  $n = d_j/d_i$ 
15      3. the first position of the matrix =  $f$ 
16    Permute the 3D matrix with the order of [2,3,1]
17  end
18  else if  $d_i > d_j$  then
19    Construct a 3D matrix, where:
20      1.  $m = d_i/d_j$ 
21      2.  $n = d_j$ 
22      3. the first position of the matrix =  $f$ 
23    Permute the 3D matrix with the order of [2,1,3]
24  end
25 end

```

6.3 Analytical Models

To discuss the cost of matrix transpose/permutation in order to eliminate uncoalesced global memory access patterns, it has to be considered whether the data size is known (predetermined) or unknown before the GPGPU run-time.

6.3.1 Predetermined Data Size

In case the data size is predetermined before the run-time, the proposed technique can be done during the compile-time on the CPU. In the case of a single

stride access pattern, an algorithm of the 2D matrix transpose as shown by the algorithm in Listing 6.12 yields the complexity of $O(nm)$.

Listing 6.12: An algorithm for 2D matrix transposition on a CPU

```

1 for  $x = 0; x < m; ++x$  do
2   for  $y = 0; y < n; ++y$  do
3      $A.t[y][x] = A[x][y];$ 
4   end
5 end

```

In the case of a double stride access pattern, a 3D matrix permutation algorithm on a CPU with the order of [2,3,1] is given by Listing 6.13. Therefore, the complexity of a 3D matrix permutation is $O(nmp)$.

Listing 6.13: An algorithm for 3D matrix permutation on a CPU with the order of [2,3,1]

```

1 for  $x = 0; x < m; ++x$  do
2   for  $y = 0; y < n; ++y$  do
3     for  $z = 0; z < p; ++z$  do
4        $A.p[y][z][x] = A[x][y][z];$ 
5     end
6   end
7 end

```

In summary, if the data size is predetermined, the complexity of the compile-time will increase by $O(nm)$ for a single stride access pattern, and $O(nmp)$ for a double stride access pattern. It is possible that n , m , and p will multiply the matrix transformation overhead, but the transformation is done only once. Therefore, the break-even point between the performance gain from the matrix transformation and the overhead have to be considered.

6.3.2 Unknown Data Size

If the data size is not predetermined before the run-time, the proposed techniques have to be applied during the run-time. This section describes the cost analysis to determine the break-even point and benefit of the proposed technique.

6.3.2.1 Single Stride

For the single stride access pattern, the 2D matrix transpose is performed to eliminate the uncoalesced global memory access. An optimized matrix transpose program is illustrated by Figure 6.8 and given in CUDA SDK [30].

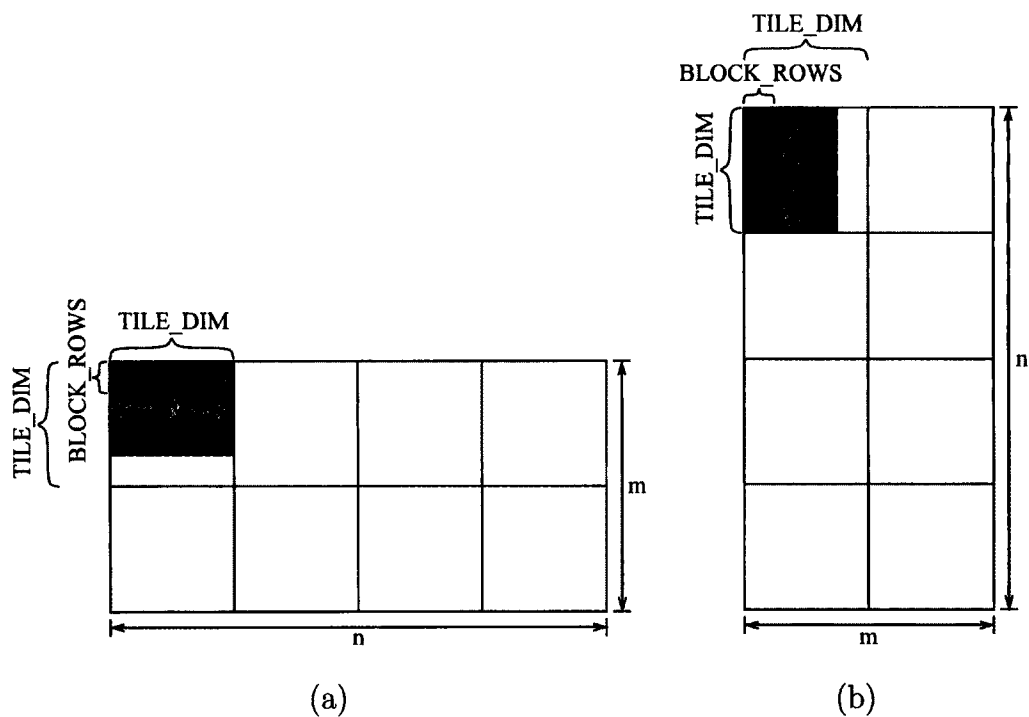


Figure 6.8: Tiled 2D matrix transposition where (a) is the original matrix and (b) is the transposed matrix

In the performance models in the previous work, which presents the mathematical models for estimating the GPU run-time with the effects of many parameters and characteristics [40], the kernel execution time depends on the following factors:

- The grid size, $S_G = \frac{nm}{\text{TILE_DIM}^2}$, where n and m are the width and height of the matrix, respectively. In addition, the shared memory size is $\text{TILE_DIM} \times \text{TILE_DIM}$.
- The block size, $S_B = \text{TILE_DIM} \times \text{BLOCK_ROWS}$, where TILE_DIM is an integral multiple of BLOCK_ROWS . Furthermore, BLOCK_ROWS is an integral multiple of the warp size, S_W . Therefore, each thread handles $\frac{\text{TILE_DIM}}{\text{BLOCK_ROWS}}$ elements.
- The memory latency in the kernel,

$$T_M = \frac{\text{TILE_DIM}}{\text{BLOCK_ROWS}} (\text{Coal_read} + \text{Smem_write} + \text{Coal_write} + \text{Smem_read}),$$

where *Coal_read* is the time that a warp reads the data from global memory coalescedly, and *Coal_write* is the time that a warp writes the data to global memory coalescedly. Moreover, *Smem_read* is the time a warp reads the data from shared memory, and *Smem_write* is the time a warp writes the data to shared memory.

- The computing time in the kernel, T_C is a function of $\frac{\text{TILE_DIM}}{\text{BLOCK_ROWS}}$.

Mostly, this matrix transpose program spends more time to move data in and out of global and shared memory. Therefore, the kernel execution time of the matrix transpose, T_{trans} , is a function of memory accesses, TILE_DIM , BLOCK_DIM , and the matrix size as described by Equation (6.1).

$$T_{trans} \approx \frac{nm}{N_{SM}N_{RB}} \cdot \frac{\text{Coal_read} + \text{Coal_write} + \text{Smem_read} + \text{Smem_write}}{\text{TILE_DIM} \times \text{BLOCK_ROWS}}. \quad (6.1)$$

6.3.2.2 Double Strides

The double stride access pattern can be eliminated by 3D matrix permutation, which can be considered in two cases: permuting the matrix by the order of [2,1,3] and permuting the matrix by the order of [2,3,1]. For permuting the matrix from [1,2,3] to [2,1,3] the third dimension is fixed. Hence, the 2D matrix transpose can be applied to each slice along the third dimension [41]. Therefore, T_{trans} can be approximated by Equation (6.2).

$$T_{trans} \approx \frac{nmp}{N_{SM}N_{RB}} \cdot \frac{Coal_read + Coal_write + Smem_read + Smem_write}{TILE_DIM \times BLOCK_ROWS}. \quad (6.2)$$

However, for permuting the matrix from [1,2,3] to [2,3,1], the grid size can be approximately determined by $S_G \approx \frac{n}{TILE_DIM^2} (p + TILE_DIM - 1)(m + TILE_DIM - 1)$. Thus, T_{trans} can be approximated by Equation (6.3).

$$T_{trans} \approx \frac{nmp(TILE_DIM)}{N_{SM}N_{RB}(BLOCK_ROWS)} \cdot (Coal_read + Coal_write + Smem_read + Smem_write). \quad (6.3)$$

6.4 Results and Cost Analysis

In this section, the effects of uncoalesced memory access are studied based on the proposed technique using sample code in Listing 6.5. They are analyzed in two cases: when the data sized is known and unknown before the run-time. However, in the double stride case, the outer-loop can be distributed across thread blocks, the analytical results of single and double stride cases are not significantly different.

6.4.1 Predetermined Data Size

Assuming that the data size is known before the run-time, the matrix transpose is done during compile-time and is not taken into account here. The results are shown in Table 6.1. It is obvious that the kernel execution time with the original memory access is longer than the kernel execution time with the access to the transformed memory alignment. For instance, to access 2^{18} elements, the original memory latency is 0.03 ms longer than the transformed memory latency. Additionally, the original memory access to 2^{20} elements access takes 0.09 ms longer than the transformed memory access. Thus, the performance gain is on average over 25 percent.

Table 6.1: The comparison between the kernel execution times based on our technique using the sample code in Listing 6.4 with the original memory access and transformed memory access.

| Count (elements) | Original memory latency (ms) | Transformed memory latency (ms) | Performance gain (%) |
|------------------|------------------------------|---------------------------------|----------------------|
| 64K | 0.03 | 0.02 | 33.3 |
| 128K | 0.04 | 0.03 | 25.0 |
| 256K | 0.08 | 0.05 | 37.5 |
| 512K | 0.14 | 0.10 | 28.6 |
| 1M | 0.27 | 0.18 | 33.3 |

6.4.2 Unknown Data Size and Break-even Analysis

In the case that the data size is unknown before the run-time, the time of matrix transpose has to be considered. Table 6.2 shows the effect of transformed memory access using 2D matrix transpose. It is seen that the 2D matrix transpose (Run-time overhead) takes longer than the time gained from transformed memory access. Thus, this technique will not be beneficial if the transformed memory is

accessed only once or twice. However, it can be beneficial if the transformed memory is accessed frequently.

For example, to access 2^{18} elements, the 2D matrix transpose takes 0.091 ms, which is over 3 times the performance gain from the transformed memory access. Hence, the proposed technique is beneficial if the transposed matrix is accessed at least 4 times.

Table 6.2: The comparison between the performance gain from the transformed memory access to the time of 2D matrix transpose

| Count (elements) | Performance gain from the transformed memory access (ms) | Run-time overhead (ms) |
|---------------------|---|---------------------------|
| 64K | 0.01 | 0.028 |
| 128K | 0.01 | 0.048 |
| 256K | 0.03 | 0.091 |
| 512K | 0.04 | 0.173 |
| 1M | 0.09 | 0.340 |

6.5 Conclusion

Optimization is a challenging problem. Uncoalesced memory access is one of the issues that decreases the performance of GPGPU applications. To reduce memory latency due to uncoalesced memory access patterns, the memory rearrangement techniques using matrix transpose/permutation are proposed. The patterns are categorized by the number of strides. For a single stride memory access pattern, a 2D matrix associated with the stride is constructed. Then 2D matrix is transposed to rearrange the data in global memory. For a double stride memory access pattern, a 3D matrix is constructed and 3D matrix permutation is performed. The analytical models have been discussed, considering whether the data size is predetermined. If

the data size is known before the run-time, the proposed techniques can be applied during the compile-time. Otherwise, the proposed techniques have to be performed at the run-time. Our analytical results have shown the performance gain and the break-even point for the proposed techniques.

Since the proposed techniques can be the most advantageous when the data size is predetermined, the future work is to predict the data size required by a GPU kernel execution and detect memory uncoalescing that may occur during the kernel execution.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

Due to the rapid increase in computational performance required to handle massive data sets, GPUs have been deployed in several HPC systems. GPUs can provide highly parallel computation via several multithreaded processors. However, there are a few studies that give an insight on both GPGPU performance and reliability.

This dissertation has presented streamed checkpoint/restart (CPR) protocols for GPGPU that utilize CUDA stream to reduce the checkpoint and recovery costs in order to minimize the total wasted time. Furthermore, the study has revealed that even though the costs can be reduced, streamed CPR may not be beneficial in a long-running application since the wasted time is dominated by the recomputing time. Consequently, a checkpoint scheduling model to minimize the recomputing time has also been proposed.

In order to derive effective checkpoint scheduling models, an application run-time is required. Therefore, a performance model for predicting a GPGPU application run-time has been proposed in this dissertation. This performance model improves the kernel execution time estimation by considering different memory access types and the impacts of other factors such as synchronization functions, branch divergence, and bank conflicts. The results have shown that this performance model can achieve

higher accuracy when the data size is predetermined and especially, practically large enough.

Furthermore, in this dissertation, the impacts of application performance and system reliability on a heterogeneous HPC system have been studied. Due to the fact that the application performance increases with the number of nodes while the system reliability decreases, the models to determine an optimal number of nodes have been proposed. These models are created based on three different criteria: maximal reliability, the desired application performance with a fixed problem size and a scalable problem size. In addition, the analytical results have indicated that the checkpoint/restart mechanism based on the proposed checkpoint scheduling model can increase the application performance when failures occur.

Moreover, this dissertation has presented an optimization technique to reduce memory latency caused by uncoalesced memory access patterns. This technique uses matrix transformation to rearrange data resided in the GPU's global memory. The optimization overhead can be estimated by the proposed analytical model. The analytical results have shown that this technique can be beneficial when the data size is predetermined or the transformed data are frequently accessed.

In the future, we hope that the models proposed in this dissertation will be implemented at compiler level to achieve automatic checkpoint and optimization tools at compile-time.

BIBLIOGRAPHY

- [1] Top 500 Supercomputing Sites. <http://www.top500.org>. Online: Accessed: Dec, 2012.
- [2] General-Purpose Computation on Graphics Hardware. <http://gpgpu.org>. Online; accessed: Dec, 2012.
- [3] NVIDIA. CUDA C Programming Guide Version 4.0, May 2011.
- [4] S. Laosooksathit, A. Baggag, and C. Chandler. Stream Experiments: Toward Latency Hiding in GPGPU. In *Proceedings of the 9th IASTED International Conference*, volume 676, page 240, 2009.
- [5] Hong Ong, Natthapol Saragol, Kasidit Chanchio, and Chokchai Leangsuksun. VCCP: A Transparent, Coordinated Checkpointing System for Virtualization-based Cluster Computing. In *IEEE Cluster*, 2009.
- [6] Hiroyuki Takizawa, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. CheCUDA: A Checkpoint/Restart Tool for CUDA Applications. In *PDCAT*, pages 408–413, 2009.
- [7] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, September 1974.
- [8] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics: Conference Series*, 46(1):494, 2006.

- [9] Adam J. Ferrari, Stephen J. Chapin, and Andrew S. Grimshaw. Process introspection: A heterogeneous checkpoint/restart mechanism based on automatic code modification. Technical report, Charlottesville, VA, USA, 1997.
- [10] X. Xu, Y. Lin, T. Tang, and Y. Lin. HiAL-Ckpt: a hierarchical application-level checkpointing for CPU-GPU hybrid systems. In *5th International Conference on Computer Science and Education (ICCSE)*, pages 1895–1899, 2010.
- [11] S. Laosooksathit, N. Naksinehaboon, C. Leangsuksun, A. Dhungana, C. Chandler, K. Chanchio, and A. Farbin. Lightweight Checkpoint Mechanism and Modeling in GPGPU Environment. *Computing (HPC systems)*, 12:13, 2010.
- [12] Yudan Liu, Raja Nassar, Chokchai Leangsuksun, Nichamon Naksinehaboon, Mihaela Paun, and Stephen L. Scott. An optimal checkpoint/restart model for a large scale high performance computing system. In *IPDPS*. IEEE, 2008.
- [13] Mihaela Paun, Nichamon Naksinehaboon, Raja Nassar, Chokchai Leangsuksun, Stephen L. Scott, and Narate Taerat. Incremental Checkpoint Schemes for Weibull Failure Distribution. *International Journal of Foundations of Computer Science*, 21(03):329, 2010.
- [14] N.R. Gottumukkala, R. Nassar, M. Paun, C.B. Leangsuksun, and S.L. Scott. Reliability of a System of k Nodes for High Performance Computing Applications. *Reliability, IEEE Transactions*, 59(1):162–169, 2010.
- [15] Thanadech Thanakornworakij, Raja Nassar, Chokchai Leangsuksun, and Mihaela Paun. Reliability Model of a System of k Nodes with Simultaneous Failures for High Performance Computing Applications. *accepted at the International Journal of High Performance Computing Applications*, Oct 2012.

- [16] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37:152–163, June 2009.
- [17] Kishore Kothapalli, Rishabh Mukherjee, Suhail Rehman, Suryakant Patidar, PJ Narayanan, and Kannan Srinathan. A performance prediction model for the CUDA GPGPU platform. In *HiPC*, pages 463–472, Kochi, India, December 2009. IEEE.
- [18] Yao Zhang and John D. Owens. A Quantitative Performance Analysis Model for GPU Architectures. In *Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA 17)*, pages 382–393. IEEE, February 2011.
- [19] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An Adaptive Performance Modeling Tool for GPU Architectures. pages 105–114, 2010.
- [20] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 235–246, March 2010.
- [21] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

- [22] James W. Demmel, Michael T. Heath, and Henk A. van der Vorst. Parallel Numerical Linear Algebra. Technical Report UCB/CSD-92-703, EECS Department, University of California, Berkeley, Oct 1992.
- [23] David Bau, Induprakas Kodukula, Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Solving alignment using elementary linear algebra. In *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computers*, pages 46–60. Springer-Verlag, 1994.
- [24] Weng-Long Chang, Jih-Woei Huang, and Chih-Ping Chu. Using elementary linear algebra to solve data alignment for arrays with linear or quadratic references. volume 15, pages 28–39, Piscataway, NJ, USA, January 2004. IEEE Press.
- [25] Ann Chervenak, Ewa Deelman, Miron Livny, Mei-Hui Su, Rob Schuler, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. Data Placement for Scientific Applications in Distributed Environments. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing, GRID '07*, pages 267–274, Washington, DC, USA, 2007. IEEE Computer Society.
- [26] Kavitha Ranganathan and Ian Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing, HPDC '02*, page 352, Washington, DC, USA, 2002. IEEE Computer Society.
- [27] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *PLDI*, pages 86–97, 2010.

- [28] Michael Bader, Hans-Joachim Bungartz, Dheevatsa Mudigere, Srihari Narasimhan, and Babu Narayanan. Fast GPGPU Data Rearrangement Kernels using CUDA. volume abs/1011.3583, 2010.
- [29] Hidehiko Masuhara, Satoshi Matsuoka, and Akinori Yonezawa. Implementing parallel language constructs using a reflective object-oriented language, 1996.
- [30] NVIDIA. online. http://www.nvidia.com/object/cuda_home_new.html.
- [31] S. Laosooksathit, N. Naksinehaboon, and C. Leangsuksan. Two-level checkpoint/restart modeling for GPGPU. In *2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*, pages 276–283. IEEE, December 2011.
- [32] NVIDIA. CUDA C Best Practices Guide, May 2011.
- [33] NVIDIA. PTX: Parallel Thread Execution ISA Version 2.1, April 2010.
- [34] Chapter 3 CUDA threads.
- [35] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40*, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.
- [36] Blaise Barney. Introduction to Parallel Computing. Online; Accessed: Jan, 2013. <https://computing.llnl.gov/tutorials/parallel.comp/>.
- [37] Mark D. Hill and Michael R. Marty. Amdahl’s Law in the Multicore Era. In *IEEE Computer Society*, pages 33–38, July 2008. http://www.cs.wisc.edu/multifacet/papers/ieeecomputer08.amdahl_multicore.pdf.

- [38] John L. Gustafson. Reevaluating Amdahl's Law. volume 31, pages 532–533, 1988.
- [39] John L. Gustafson, Gary R. Montry, Robert E. Benner, C. W. Gear, John L. Gustafson, Gary R. Montry, Robert, and E. Benner. Development of Parallel Methods for a 1024-Processor Hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9:609–638, 1988.
- [40] S. Laosooksathit and C. Leangsuksun. A Performance Model for Predicting Completion-time of a GPGPU Application. Technical report, 2012. Unpublished manuscript.
- [41] Lung-Sheng Chien. Matrix transpose.
http://oz.nthu.edu.tw/d947207/index_3Ddata.htm.