COMPILER-MANAGED FAULT TOLERANCE TECHNIQUES FOR GENERAL PURPOSE GRAPHICS PROCESSING UNITS

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Ercüment KAYA

June 2022
İZMİR

ACKNOWLEDGMENTS

First, I would like to thank my supervisor, Dr. Işıl Öz. Her patience and trust made this thesis possible. Her dedication and respect for her students inspired me to better myself.

We, as PARS Lab, would like to thank the Scientific and Technological Research Council of Turkey (TÜBİTAK) for supporting our work (Grant No: 119E011), and we also would like to thank CERCIRAS COST Action CA19135 funded by COST Association to publish and partially support our work.

I would like to thank my colleagues from PARS Lab Emre Dinçer and Burak Topçu. I also thank my colleagues Efe Gürman and Uğur Deniz Yüksel for their understanding and support.

I would like to thank my parents, Şaban and Müberra. Their support from my childhood to this day made my every achievement possible. I would not be able to write this thesis without their unconditional support and love. I cannot thank them enough. I also would like to thank my little sister, Beyza Nur, for her odd sense of humor.

I wish to thank my two precious cats: Bergen and Miniş. Two years ago, they were stray cats. Miniş was the only survivor of his mother's birth. He got sick after a while. One day, I saw him sick and took him to a veterinarian. The veterinarian said that he might not get through the night. Yet, he got better. Now he is one of the smartest I know. Whenever I feel blue, he licks my hair. He is a fighter that inspired me. On the other hand, Bergen is one of the most affectionate cats I know. Every time I go by him, he purrs and even hugs me. Sadly, our veterinarian had to pull his six teeth. Yet, he got through. To all the people that wish for a pet, please adopt a stray animal.

Lastly, but not least, I would like to express sincere appreciation to my soon-to-be wife, Ferda Gül Palalı. She is an unshakable source of support, understanding, and love. Her unwavering love is the source of my self-esteem. There is no word to describe her importance in my life. I am more than sure that every problem I face and I will face, gets easy with her by my side. I cannot wait to spend my life with her. This thesis could not be possible without her.

For everything in the world, for civilization, for life, for success, the truest guide is science.

–Gazi Mustafa Kemal Atatürk Paşa

ABSTRACT

COMPILER-MANAGED FAULT TOLERANCE TECHNIQUES FOR GENERAL PURPOSE GRAPHICS PROCESSING UNITS

As the use of graphics processing units evolves for general-purpose computations besides inherently-fault tolerant graphics programs, soft error reliability becomes a first-class citizen in program design. In this thesis, we aim to increase the reliability of general-purpose graphics processing units.

We propose compiler-based redundancy schemes for graphics processing units. Our framework replicates the annotated kernel function by a programmer at compile time. Our selective redundancy approach enables us to provide full redundancy with no error and partial redundancy with an acceptable error rate with higher performance. We develop different schemes to satisfy the performance and memory requirements of the general-purpose graphics processing unit applications.

We build our framework on top of the LLVM compiler framework to increase the reliability of applications that exploit the CUDA programming model and evaluate our schemes for the applications from the PolyBench benchmark suite. We reveal that our partial redundancy approach improves the reliability with a small performance overhead and our full redundancy schemes provide complete fault coverage with varying performance differences based on the application's characteristics.

ÖZET

GENEL AMAÇLI GRAFIK İŞLEME BIRIMLERI İÇIN DERLEYICI TARAFINDAN YÖNETILEN HATA TOLERANS TEKNIKLERI

Grafik işlem birimleri, doğası gereği hataya dayanıklı grafik programlarının yanında genel amaçlı hesaplamalar için kullanılması artmasından dolayı, yumuşak hata güvenilirliği, program tasarımında önemli bir sorun haline gelir. Bu tezde, genel amaçlı grafik işlemci birimlerinin güvenilirliğini artırmayı hedefliyoruz.

Bu çalışmada, grafik işleme birimleri için derleyici tabanlı seçici yedekliliği öneriyoruz. Çerçevemiz, programlamacı tarafından işaretlenmiş çekirdek işlevini derleme zamanında çoğaltır. Seçici yedeklilik yaklaşımımız, hatasız tam yedeklilik ve daha yüksek performansla kabul edilebilir bir hata oranıyla kısmi yedeklilik sağlamamızı sağlar. GPGPU uygulamalarının performans ve bellek gereksinimlerini karşılamak için farklı şemalar geliştiriyoruz.

Çerçevemizi LLVM derleyici çerçevesinin üzerine kurduk, programlama modeli olarak CUDA programlama modelini; şemalarımızı değerlendirmek için PolyBench kıyaslama uygulamalarını kullanıyoruz. Kısmi yedeklilik yaklaşımımızın, küçük bir performans gecikmesi ile güvenilirliği geliştirdiğini ve tam yedeklilik şemalarımızın, uygulamanın özelliklerine bağlı olarak değişen performans farklılıklarıyla eksiksiz hata kapsamı sağladığını ortaya koyuyoruz.

TABLE OF CONTENTS

LIST OF TABLES viii

LIST OF FIGURES ix

LIST OF LISTINGS xi

CHAPTER 1. INTRODUCTION 1

1.1. Contributions of Thesis 2

1.2. Organization of Thesis 3

CHAPTER 2. BACKGROUND AND MOTIVATION 4

2.1. GPU Architecture and Programming Model 4

2.2. Soft Error Reliability in GPGPUs 7

2.3. Redundant Multithreading 8

2.4. Compilers 9

CHAPTER 3. RELATED WORKS 11

3.1. Hardware and Architecture-Based Techniques 11

3.2. Compiler Based Techniques 13

3.3. Application Based Techniques 16

CHAPTER 4. PROPOSED FRAMEWORK 17

4.1. Fault Model 17

4.2. Fault Injection Experiments 19

4.3. Custom Compiler 20

4.3.1. Code Annotation 20

4.3.2. Output Replication 24

4.3.3. Redundant Kernel Execution 25

4.3.3.1. Memory Allocation Techniques 25

4.3.3.1.1 Large Memory 26
4.3.3.1.2 Small Memory 28
4.3.3.2. Execution Techniques 29
4.3.3.2.1 Multiple Kernel Execution (MKE) 29
4.3.3.2.2 Multiple Kernel Executions with Stream Enabled (MKES) 30
4.3.3.2.3 Single Kernel Execution (SKE) 31
4.3.4. Output Comparison 36

CHAPTER 5. EXPERIMENTAL RESULTS 38
5.1. Experimental Setup 38
5.2. Experimental Results.. 38

CHAPTER 6. CONCLUSION AND FUTURE WORKS 65
6.1. Future Works 66

REFERENCES 67

LIST OF TABLES

Table **Page**

Table 4.1 ThreadID Modification Codes for Our SKE-based Schemes. 35

Table 5.1 Salient Characteristics of GPU Devices Used in Our Experiments. ... 39

Table 5.2 Execution times for the kernel functions. 41

Table 5.3 Configurations of 2D Convolution. 46

Table 5.4 Configurations of Correlation, Covariance, and Gemm 49

Table 5.5 Configurations of 3D Convolution, Fdtd2d and Gramschmidt 55

Table 5.6 Configurations of 3mm and Bicg 60

LIST OF FIGURES

Figure **Page**

Figure 2.1 Grid and Block Organization in a CUDA Kernel. 5
Figure 2.2 Error Classification (Source: Mukherjee (2011)). 7
Figure 2.3 Major LLVM Components (Source: Lattner (2012)). 9
Figure 4.1 General Diagram of Our Compiler Framework. 18
Figure 4.2 Diagram of the Regional Fault Injection Tool (Source: Öz and Karadaş, (2022)). 20
Figure 4.3 Valid Redundancy Schemes. 22
Figure 4.4 General Form of Large Memory Based Techniques. 27
Figure 4.5 General Form of Small Memory-Based Techniques. 28
Figure 4.6 Grid Organizations for Original Execution. 32
Figure 4.7 Grid Organizations for LX*SKE Schemes. 33
Figure 4.8 Grid Organizations for LY*SKE 34
Figure 5.1 SDC Rates for the Kernel Functions. 40
Figure 5.2 Normalized Execution Times for the Redundant Executions. 42
Figure 5.3 The Change in Percentage of SDC Rates and Execution Times for the Redundant Executions. 43
Figure 5.4 Execution Time Profile of Each Function. 44
Figure 5.5 Normalized Execution Times of 2D Convolution with Large Memory Schemes in Quadro P620 and Tesla K80. 47
Figure 5.6 Normalized Execution Times of 2D Convolution with Small Memory Schemes in Quadro P620 and Tesla K80. 48
Figure 5.7 Normalized Execution Times of Correlation, Covariance, and Gemm with Large Memory Schemes in Quadro P620. 50
Figure 5.8 Normalized Execution Times of Correlation, Covariance, and Gemm with Large Memory Schemes in Tesla K80. 51
Figure 5.9 Normalized Execution Times of Correlation, Covariance, and Gemm with Small Memory Schemes in Quadro P620. 53
Figure 5.10 Normalized Execution Times of Correlation, Covariance, and Gemm with Small Memory Schemes in Tesla K80. 5

Figure		**Page**

Figure 5.11 Normalized Execution Times of 3D Convolution, Fdtd2d, and Gram-schmidt with Large Memory Schemes in Quadro P620. 56

Figure 5.12 Normalized Execution Times of 3D Convolution, Fdtd2d, and Gram-schmidt with Large Memory Schemes in Tesla K80. 57

Figure 5.13 Normalized Execution Times of 3D Convolution, Fdtd2d, and Gram-schmidt with Small Memory Schemes in Quadro P620. 58

Figure 5.14 Normalized Execution Times of 3D Convolution, Fdtd2d, and Gram-schmidt with Small Memory Schemes in Tesla K80. 59

Figure 5.15 Normalized Execution Times of 3mm and Bicg with Large Memory Schemes in Quadro P620. 61

Figure 5.16 Normalized Execution Times of 3mm, and Bicg with Large Memory Schemes in Tesla K80. 61

Figure 5.17 Normalized Execution Times of 3mm and Bicg with Small Memory Schemes in Quadro P620. 63

Figure 5.18 Normalized Execution Times of 3mm, and Bicg with Small Memory Schemes in Tesla K80. 64

LIST OF LISTINGS

Listing **Page**

Listing 2.1 Typical CUDA Application..6
Listing 2.2 Example RMT and Majority Voting Example Implementation.............8
Listing 4.1 Annotated Initial *Fdtd2d* Code. 23
Listing 4.2 Generated LLVM IR Code...23
Listing 4.3 CUDA Code for Large Memory Based Technique. 27
Listing 4.4 CUDA Code for LMKE Scheme. 30
Listing 4.5 CUDA Code for LMKES Scheme. 30
Listing 4.6 Output Identification Code...36
Listing 4.7 Inspired Detection Function...36
Listing 4.8 Inspired Majority Voting Function...37

CHAPTER 1

INTRODUCTION

As the impact of computer systems in our lives grows, the need for higher performance and lower resource consumption increases. To fulfill this requirement, heterogeneous computing systems are built. They contain multiple computational units and offer higher performance with less resource consumption by combining different devices.

Heterogeneous computing systems can be built using general-purpose multi-core processors (CPUs) and data-parallel graphic processing units (GPUs). Those systems provide efficient computation for high computational performance and less resource consumption in large-scale computing platforms. Over the years, GPU architectures with high computation power and massively parallel architecture have been widely utilized for general-purpose computations as well as graphics applications (Lee et al. (2010); Kirk and Wen-Mei (2016); Mittal and Vetter (2014); Leng et al. (2013); Aamodt et al. (2018)).

Soft errors are a subset of transient faults. Their leading cause is a temporary malfunctioning of the system. Various environmental factors such as radiation and internal conditions like electrical noise generated by components may cause a bit flip that leads to erroneous results (Kim et al. (2014)).

The domain change of general-purpose GPUs makes error vulnerability a significant concern. Even though graphics applications are inherently error-tolerant, many domains such as medicine and defense do not have such a characteristic. Therefore, error tolerance in general-purpose GPUs has significant importance.

Over the years, various fault tolerance techniques on different devices and levels have been developed to deal with hardware errors. Hardware-based techniques such as error correction codes (ECC) provide extremely reliable execution with additional hardware costs. On the other hand, software-based techniques provide fault tolerance by executing redundant copies of target code. However, they cause performance latency. Redundant multithreading (RMT) is one of the widely used techniques. This technique can be employed on instruction, function, and application levels. Instruction-level approaches pro-

vide high-level fault tolerance by replicating instructions (Reis et al. (2005), Bohman et al. (2019), Didehban and Shrivastava (2016)). Due to its fine-grained nature, instruction-level approaches increase code size rapidly, which may lead to overhead. Compiler-level approaches can be achieved with hand-coded solutions. However, compilers automate the procedure and create the executable without additional intervention. Function-level approaches replicate functions instead of instruction. Its coarse-grained nature does not increase the code size significantly.

Many of the past works execute the application with full redundancy. Only a small part of the work offers partial redundancy. Full redundancy offers fully fault coverage. However, it may result in inefficiency of resource usage as well as performance loss. On the other hand, partial redundancy offers acceptable error rates and lower overhead than full redundancy. It provides more efficient resource usage and less performance loss.

Generally, partial redundancy is provided using either a heuristic or an annotation. The framework determines the regions (such as instructions) to replicate if a heuristic is used. In this way, the application's reliability increases without intervention. On the other hand, if an annotation is used, the programmer marks the will-to-be-replicated regions such as functions. Even though this way requires an additional alteration in the source code, it gives more control to the programmer.

1.1. Contributions of Thesis

This work's main contributions are listed below:

- We build a compiler-managed fault tolerance framework. Our framework increases the reliability of the programmer-selected functions. We perform fault injection experiments and examine the target kernels. Based on the results, we implement a partial redundancy approach for GPU applications by only replicating more error-prone regions that perform better compared to full redundant applications. The initial version of our work has been presented at the 1st Workshop on Connecting Education and Research Communities for an Innovative Resource Aware Society (CERCIRAS) (Kaya et al. (2021)).

- We consider various features to exploit the massively parallel architecture of GPUs to reduce the overhead of redundant executions.
- We implement different redundancy approaches by considering different performance and memory requirements and examining their characteristics in different applications and input sizes.

1.2. Organization of Thesis

The thesis is outlined as follows. Chapter 2 provides background information about GPU architecture, compilers, and our motivations. Chapter 3 presents related work in fault tolerance. Chapter 4 presents our approaches in detail. Chapter 5 demonstrates our experimental results. Lastly, Chapter 6 contains conclusions and future works.

CHAPTER 2

BACKGROUND AND MOTIVATION

This chapter gives essential information about GPU architecture, CUDA programming model, soft error reliability in GPGPUs, redundant multithreading, and compilers.

2.1. GPU Architecture and Programming Model

Initially, GPU devices have been used for rendering graphics applications in real-time. However, currently, they also support non-graphics applications. GPUs follow the Single Instruction Multiple Data (SIMD) execution model, where processing units execute the same instruction with different data (Aamodt et al. (2018)). Therefore, they are beneficial for large-scale applications and data-level parallelism, especially in domains such as machine learning and computer vision (Schlegel (2015), HajiRassouliha et al. (2018)).

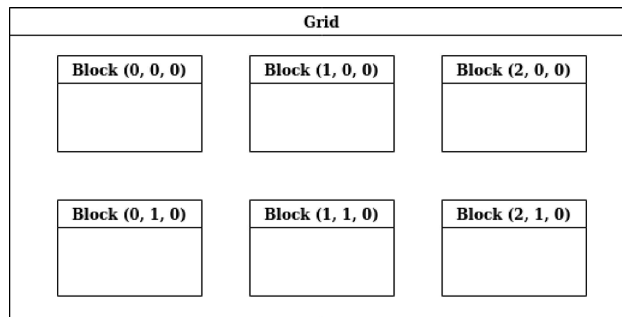
GPU devices have a large number of processing units. Therefore, they can execute with many threads in a given time. This feature allows programmers to execute the applications in massively parallel and with higher performance.

CUDA is a programming model developed by NVIDIA for NVIDIA GPUs (Kirk and Wen-Mei (2016)). A typical CUDA program [1] as seen in the Listing 2.1 begins its execution in a CPU (Host), allocates memory space on GPU (Device) (lines 32, 36, 40), transfers data from host to device (lines 42, 43), and launches a kernel function execution by creating multiple threads (line 47).

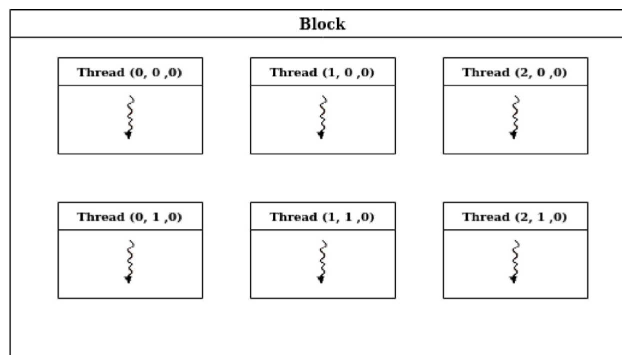
A *thread* is the smallest execution unit of a CUDA application. Multiple, usually 32, threads create a scheduling unit called *warp*. CUDA kernel functions operate with multiple threads in a gradual thread distribution. Each kernel function is operated as a *grid* of blocks, and each *block* consists of multiple threads. Both grids and blocks have

[1] Taken from https://github.com/nvidia/cuda-samples

various dimensions, as shown in Figure 2.1. This feature has advantages for working with multi-dimensional data. The figures show the two-dimensional layout of blocks and grids. Since the third dimension is not widely used and we do not utilize it, we do not depict it.



(a) An Example Grid.



(b) An Example Block.

Figure 2.1. Grid and Block Organization in a CUDA Kernel.

```
1  #include <stdio.h>
2  #include <cuda_runtime.h>
3  #include <helper_cuda.h>
4  /* CUDA Kernel Device code */
5  __global__ void vectorAdd(const float *A, const float *B, float *C,
6                            int numElements) {
7    int i = blockDim.x * blockIdx.x + threadIdx.x;
8
9    if (i < numElements) {
10     C[i] = A[i] + B[i] + 0.0f;
11   }
12 }
13
14 /* Host main routine */
15 int main(void) {
16
17   int numElements = 50000;
18   size_t size = numElements * sizeof(float);
19
20   float *h_A = (float *)malloc(size);
21   float *h_B = (float *)malloc(size);
22   float *h_C = (float *)malloc(size);
23
24   // Initialize the host input vectors
25   for (int i = 0; i < numElements; ++i) {
26     h_A[i] = rand() / (float)RAND_MAX;
27     h_B[i] = rand() / (float)RAND_MAX;
28   }
29
30   // Allocate the device input vector A
31   float *d_A = NULL;
32   cudaMalloc((void **)&d_A, size);
33   // Allocate the device input vector B
34
35   float *d_B = NULL;
36   cudaMalloc((void **)&d_B, size);
37
38   // Allocate the device output vector C
39   float *d_C = NULL;
40   cudaMalloc((void **)&d_C, size);
41
42   cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
43   cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
44   // Launch the Vector Add CUDA Kernel
45   int threadsPerBlock = 256;
46   int blocksPerGrid = 210
47   vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
48   /* Copy the device result vector into the device memory
49     to the host result vector in host memory. */
50   cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
51   // Free device global memory
52   cudaFree(d_A);
53   cudaFree(d_B);
54   cudaFree(d_C);
55   // Free host memory
56   free(h_A);
57   free(h_B);
58   free(h_C);
59   return 0;
60 }
```

Listing 2.1 Typical CUDA Application.

2.2. Soft Error Reliability in GPGPUs

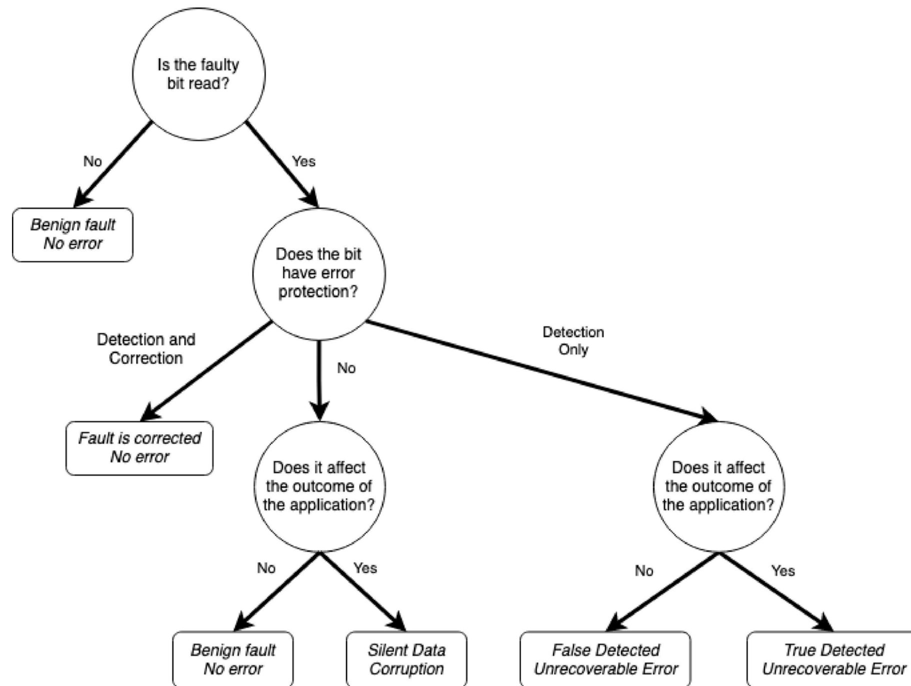


Figure 2.2. Error Classification(Source: Mukherjee (2011)).

A *fault* is defined as an inaccurate state of an application or component outcoming from failures of its units (Koren and Krishna (2020)). The cause of a fault may vary from system misuses to design errors. There are three major types of faults: permanent, intermittent, and transient. The leading cause of permanent faults is permanent physical alterations in hardware components. Intermittent faults occur occasionally and repeat themselves. Lastly, transient faults' leading cause is a temporary malfunctioning in the system.

We focus on soft errors in this thesis, a subset of transient faults. They result from single bit-flips in computer system structures and occur due to internal conditions such as generated electrical noise or environmental conditions such as cosmic rays and thermal neutrons (Mukherjee (2011)). Figure 2.2 shows error classification. If a soft error occurs in a register or memory location, it may cause corruption in data or the application crash. The fault may be masked due to being ignored by the application or corrected by any error

correction mechanism such as an error correction code (ECC). However, it may affect the outcome of the application via silent data corruption (SDC) or detected unrecoverable error (DUE). Since SDC produces incorrect outputs and applications seem to be executed successfully, SDC is the most critical fault type.

When GPUs were only used for graphics applications, they were considered inherently fault-tolerant. However, as they are increasingly used to accelerate general-purpose computations, their soft error resilience becomes a more critical issue. Therefore, the issue of soft errors for GPU systems has recently become a significant design challenge. In this work, we focus on the soft error evaluation of GPU systems.

2.3. Redundant Multithreading

Redundant multithreading is a widely used and efficient technique for dealing with soft errors (Oz and Arslan (2019)). As seen in Listing 2.2, it executes given pieces multiple times and performs a correction mechanism such as majority voting if an error occurs. From Line 3 to 7, there is an example of a majority voting function implementation. This implementation assumes that at least two of the generated output equals each other.

```
1  #include <stdio.h>
2
3  int majorityVoting(int copy, int copy_1, int copy_2){
4      if(copy != copy_1)
5          return copy_2
6      return copy;
7  }
8  void add(int A, int B, int c) {
9    c = a + b;
10 }
11 int main(void) {
12   int A = 17;
13   int B = 72;
14   int C, C_copy1, C_copy2;
15   add(A, B, C);
16   add(A, B, C_copy1);
17   add(A, B, C_copy2);
18   C = majorityVoting(C, C_copy1, C_copy2)
19   return 0;
20 }
```

Listing 2.2 Example RMT and Majority Voting Example Implementation.

Redundant multithreading (RMT) approaches have been implemented at various levels to overcome soft errors. RMT uses parallelism in parallel systems and makes fault

detection and correction possible. RMT can correct faults by executing identical application copies as discrete threads in parallel execution units and comparing executions' outputs. Parallel threads may reduce redundancy overheads by removing redundant execution latency. However, having several threads may incur additional overheads due to contention among many threads.

2.4. Compilers

A *compiler* is a software tool that converts high-level programming languages like C and C++ into machine code such as x86 and NVPTX. Clang and GNU Compiler Collection (GCC) are widely used for compiling C and C++ programs.

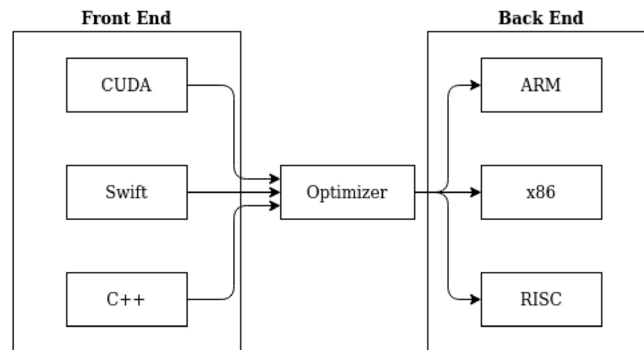


Figure 2.3. Major LLVM Components(Source: Lattner (2012)).

Clang, which we use for compiling our target applications, is based on LLVM. LLVM is a compiler framework that supports many programming languages and architectures thanks to its modular structure (Lattner and Adve (2004)). LLVM consists of three major parts, as shown in Figure 2.3. While compiling a code with LLVM, firstly, the front-end takes a language-dependent code, such as C and Swift, as input and generates its corresponding LLVM Intermediate Representation (IR) code. LLVM IR is the fundamental part of LLVM. It sits in the middle-end after the programming language but before the architecture. Therefore, it is language and architecture-independent. The syntax of LLVM IR is similar to assembly; however, it is more readable. The primary advantage of LLVM IR is that it helps to abstract source code from programming language

and architecture thanks to its language and architecture-independent nature. Hence, a programmer can create more generic optimizations. After the front-end generates the IR code, the optimizer takes an LLVM IR code as input and generates an optimized LLVM IR code. Finally, the back-end takes an LLVM IR code as input and generates a corresponding architecture-dependent machine code, such as x86 and NVPTX. Since LLVM is open-source and supports CUDA, we build our framework on top of it (LLVM Project (2022)).

CHAPTER 3

RELATED WORKS

Over the years, researchers from academia and industry have studied fault tolerance and reliability on various devices and levels. We divide this chapter into three sections based on level: hardware and architecture-based techniques, compiler-based techniques, and application-based techniques.

3.1. Hardware and Architecture-Based Techniques

Hardware and architecture-based techniques generally aim to protect caches, registers, and register files while assuming that other techniques such as error correction codes (ECC), single error correction, double error detection (SECDED) protect the main memory. The use case of the ECC is for error detection and correction in unusual environments and noisy channels. Hamming (1950) developed the first ECC called Hamming code. It uses parity bits to detect and correct errors. Since it has limited redundancy, it can work when the error rate is low. Its successor, SECDED, eliminates the limitation and is widely used in ECC Memory (Yitbarek and Austin (2018)). Hardware and architecture-based techniques can make alterations in hardware and architecture such as additional circuits and new instructions.

Kim and Somani (1999) offer a fault protection technique for *L1 Caches*. Due to L1's nature, only a small portion of data is extensively accessed. Their technique uses discrete *circuits* to offer fault protection. Provided fault protection only is applied to extensively accessed cache blocks. Compared to the traditional techniques that provide fault protection for all cache blocks, the advantage of this technique is lower overhead.

Zhang et al. (2003) present an in-cache replication approach to protect caches. They assume that a cache block is dead if it has not been used for a certain amount of time. The dead blocks are used to create replicates of protected blocks. They examine

two replication schemes: In the first scheme, a replication of a block is created when a cache miss occurs and at write-access. On the second scheme, it is created only at write-access. The latter scheme does not create read-only data, and therefore, it can create a replication of a higher amount of modified data. It provides a decent balance of reliability and performance.

Sugihara et al. (2007) propose two approaches to increase cache reliability. In the first approach, named error-detection, data of a cache way is replicated in another cache way. Therefore, by comparing their content, errors can be detected. However, this approach does not provide error correction. In the second approach, named error-correction, data of a cache way is replicated into two other cache ways. Therefore, if an error occurs in one of the cache ways, it can be corrected using the other two cache ways. The disadvantage of those approaches is reduced cache capacities.

Memik et al. (2005) notice that numerous registers are not in use, and they offer two schemes for duplicating active registers in unused registers. In the first scheme, *conservative*, replications of active registers are created in inactive registers without causing any performance bottleneck. However, the replications are not created if registers are under high pressure. In the second scheme, *aggressive*, registers not used for a while are marked as dead and used for duplicating the active registers. This scheme increases the access to register and duplicates them with a slight performance cost.

Tabkhi and Schirner (2012) reveal that general-purpose registers are error-prone in several applications since they are highly used. However, most of the special registers stay unused. They offer to duplicate the contents of error-prone registers into unused registers to improve reliability.

Yang et al. (2021) use the fact that every thread in a warp executes the same instruction at a given time. They propose a selective replication scheme by rearranging threads with the same vulnerability behavior into the same warps. Hence, unreliable threads are placed into the same warps. They notice that replication of the entire warp that contains just unreliable threads is more efficient. This work requires alterations in architecture.

SInRG (Mahmoud et al. (2018)) offers a software-based instruction replication approach for error detection in GPUs and evaluates runtime overheads with SDC reduc-

tion. It is neither selective based nor uses heuristics to determine which instruction to replicate. SInRG replicates all instructions in the given application. Even though it is effective, this approach causes conspicuous overhead. SInRG was implemented in NVCC, which is the out-of-box compiler for CUDA. Therefore, it gains more control over the instructions and a more efficient final executable. Along with the software part, SInRG requires specialized ISA instructions to achieve more efficiency during replication.

Even though hardware-based techniques provide high reliability, they require alterations in the hardware or are application-specific. Therefore, they are not suitable for general purpose applications and devices.

3.2. Compiler Based Techniques

Compilers are one of the essential parts of programming and computer systems. Compilers not only convert the high-level programming languages to machine codes but can also alter the application for increasing performance and reliability. This feature of compilers makes them potent tools.

Since compilers can alter the application, they are widely used to increase reliability without an additional hand-coded solution. Compilers can automate this procedure without any intervention from the programmer.

The traditional approach, SWIFT (Reis et al. (2005)), offers instruction-based redundancy to provide high-level protection. Unlike prior works, it is a software-based approach and requires no hardware alteration except ECC in the memory subsystem. SWIFT does not require double the memory by assuming the utilization of ECC in caches and memory. SWIFT uses compiler-based transformation that replicates the instructions in an application and inserts comparison instructions during code generation. The disadvantage of SWIFT is that it cannot fully protect branches.

Further research called nZDC (Didehban and Shrivastava (2016)) aims to provide near-zero silent data corruption by removing the limitations of SWIFT. nZDC checks store instructions by reloading the stored value, and it also checks load, compare, and branch instructions by duplicating. Along with the signature checking, nZDC detects branches in

the wrong direction using a direction check. nZDC protects the register file by checking the shadow registers after non-duplicated instructions. The difference between our work and SWIFT/nZDC is that we replicate functions instead of instructions. In this way, we aim to reduce the overhead.

Feng et al. (2010) offer a compiler-based technique to find static instructions that might cause a user-visible fault, such as memory access exceptions. The work uses only software-based duplication. Their technique trades off full error coverage for decreased performance and overhead of duplication. Therefore, it does not fit for mission-critical systems. Unlike our techniques, that technique also replicates the instructions. Besides, this work aims to reduce user-visible faults. On the other hand, our work decrease faults whether the faults are user-visible or not.

Xu et al. (2011) propose a compiler optimization scheme to increase the reliability of register files by reordering the instructions. They use a dynamic programming strategy under the constraint of instruction dependencies to create basic-block scheduling. This work targets the error-prone register. They aim to reduce the time interval between error-prone register accesses by reordering the code. In this way, they do not offer selective replication. Hence, the overhead could be higher than the programmer can afford. We target error-prone kernel functions with selective replication. Thus, the programmer adjusts the replication based on requirements.

Tavarageri et al. (2014) offer a compiler-based approach to detect soft errors. They indicate that the variable is error-prone between a variable's definition and its uses. In their approach, a variable's definition conduces to a *definition checksum*. Every use also conduces to a *use checksum*. They track the number of uses of the variable. After the application's execution, an error has occurred if the *definition checksum* adjusted by the number of uses is not equal to *use checksum*. However, if multiple errors occur, it would likely produce the correct checksum. To solve this issue, they propose the use of multiple checksums. Unlike this work, our work not only detects the faults but also corrects them.

Wadden et al. (2014) offer compiler-level redundant multithreading schemes for OpenCL-based GPU applications. They develop three RMT schemes: Intra-Group RMT, Intra-Group RMT + Local Data Share (LDS), and Inter-Group RMT. Intra-Group RMT schemes double the size of each *work-group*. After that, it creates redundant work-item

pairs in the work-group using work-item identification numbers. Since local memory is software-managed, the work-items in a work-group share a local address space. Therefore, they include or exclude the LDS from the sphere of replication. On the other hand, Inter-Group RMT doubles the number of work-groups in the global NDRange, then assigns the redundant work-item pairs in separate work-groups based on OpenCL's work-item identification numbers. Their RMT schemes exhibit a small overhead. In this thesis, We are inspired by the techniques proposed in this work. Instead of OpenCL, we use CUDA. This work uses a production-quality $OpenCL^{TM}$ compiler, which would provide a better executable. On the other hand, we use open-source Clang/LLVM to build our framework.

Gupta et al. (2017) design compiler techniques using fingerprinting and cross-lane operations to decrease synchronization overhead among redundant instructions and offer more efficient redundant executions for both thread-based and block-based redundancy schemes. Similarly, our work also contains thread and block-based redundancy schemes. However, our work contains redundant kernel executions.

Kalra et al. (2020) propose a selective compiler-level technique named ArmorAll to deal with the effects of soft errors in GPUs. ArmorAll offers three different compiler-based redundancy schemes, which are named as follows: *Address Armor, Value Armor,* and *Hybrid Armor*. Address Armor and Value Armor protect the addresses utilized by memory instructions and the values by replicating every instruction that participates in their calculation, respectively. Hybrid Armor protects both address and value. Although ArmorAll's schemes are selective-based, it does not allow the user to select a specific code region, and its instruction-based replication scheme increases the code size. ArmorAll uses heuristics to determine which part of the application to replicate. We use annotations that the programmer provides to determine the will-to-be-replicated parts.

Bohman et al. (2019) offers a compiler-assisted software fault tolerance tool named COAST for microcontrollers that are extensively used in task-based programming and extraordinary environments. Although COAST provides selective replication based on the code annotation, which would give the programmer option to choose, its instruction-based replication strategy results in an increased code size significantly and causes a memory bottleneck. Unlike this work, our work targets GPUs on the function level instead of

microcontrollers on the instruction level.

3.3. Application Based Techniques

Domains such as defense and medicine require additional reliability due to their unconventional environment and the need for precision. Therefore, researchers have developed new techniques to increase the reliability of applications with similar requirements.

Chen et al. (2018) offer efficient algorithm-based fault tolerance (ABFT) for matrix decomposition operations and perform source-code level fault injection experiments to determine the fault tolerance capability of their technique. They utilize the relationship between the input matrix and its checksum to perform full checksum efficiently for matrix decomposition operations instead of significant overhead triple multithreading redundancy (TMR). The offered fault tolerance technique presents high fault coverage with little performance and memory overhead.

Chang et al. (2019) evaluate SDC coverage of IR-level instruction replication by comparing hardware-level redundancy approaches. They explore that only a small amount of soft errors (0.25% on average) can escape with complete IR-based redundant execution. This study supports our assumption that our IR-based compiler-level redundancy schemes enable enough protection for target program executions.

Portet et al. (2020) and Alcaide et al. (2021) present hardware and software redundancy schemes for safe execution of automotive applications running on GPU-based heterogeneous systems. In their software redundancy scheme, they utilize stream-based redundant kernel execution, and their software approach proposes modification for target programs to enable concurrent execution of redundant kernels in different streams. However, they modify target code without compiler support and do not evaluate additional parallelism techniques for redundant execution.

Application-based techniques are very fitted for target applications. However, they are not able to provide similar results for another application.

CHAPTER 4

PROPOSED FRAMEWORK

In this thesis, we explore distinct redundancy schemes by considering both serial and parallel execution of redundant copies in GPU architectures. Our annotation-based techniques can offer partial redundancy as well as full redundancy. Partial redundancy offers balancing between error rate and performance loss. Instead of replicating the whole application, we replicate most error-prone kernel functions. We perform a fault injection experiment to determine each kernel function's silent data corruption (SDC) rates in the target GPGPU program. Using the provided information, we create a redundant executable using our custom compiler and perform our experiments.

We perform experiments using the fault injection tool to determine each kernel function's silent data corruption (SDC) rates in the target GPGPU program. Our custom compiler creates the target application's executable to reduce the vulnerability of the annotated kernels.

As shown in Figure 4.1, firstly, we perform fault injection experiments. The fault injection tool generates faults to determine the vulnerabilities of each kernel of the target application. Based on the results, the programmer uses our custom compiler to improve the reliability of the application. Our custom compiler [1] has four parts: code annotation, output replication, redundant kernel execution, and output comparison. In the code annotation part, the programmer alters the source with the custom annotation to mark the will-to-be-replicated functions. In the output replication part, we replicate the output based on the annotation. In the redundant kernel execution part, we implement twelve distinct schemes to exploit massively parallel GPU architecture. In the output comparison part, we also implement two different techniques to fulfill the memory requirements that we mention in the following sections. Our custom compiler utilizes the annotations to determine which execution techniques and output comparison techniques to use.

[1] https://github.com/parsiyte/GPU-Kernel-Redundancy

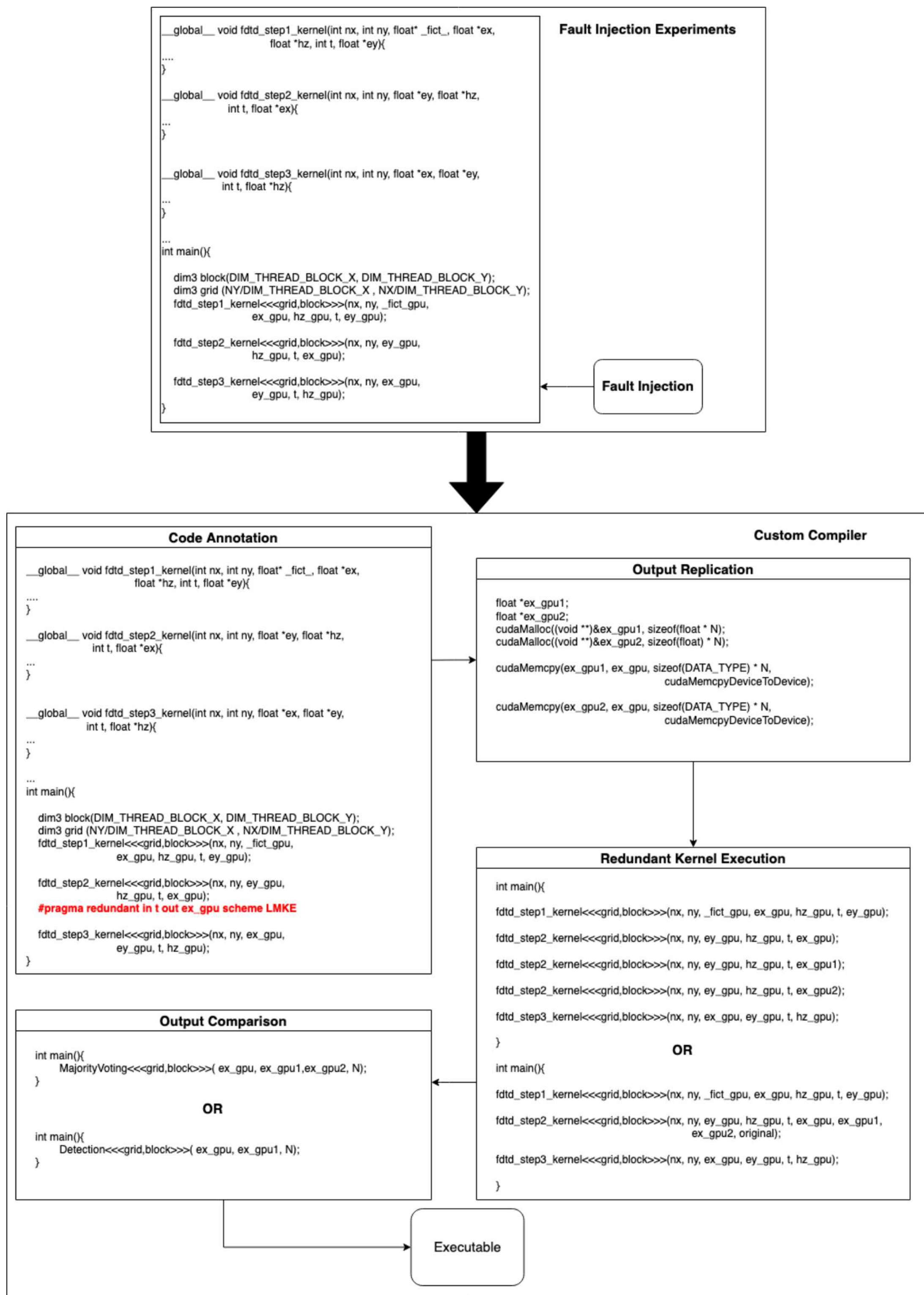


Figure 4.1. General Diagram of Our Compiler Framework.

4.1. Fault Model

Soft errors are a subset of transient faults that occur due to internal or environmental conditions resulting in bit-flip. A soft error in a register file or memory may lead to data corruption or the application crash. However, the fault may be masked due to being ignored by the application, or any error correction mechanism can correct the fault. Even though a soft error may not result in the application crash, it might cause incorrect output.

Soft errors can corrupt the output of the application. Since they do not crash the application, they are hard to detect and correct. The traditional approach for detecting soft errors is executing the application twice and comparing the results. On the other hand, to correct, we execute the application at least three times, and we decide the correct output by applying majority voting. In general, majority voting implementations assume that at least two of the three outputs are equal.

A region of the application might be more error-prone than the rest, and some domains, such as image processing, can afford slight erroneous output. Instead of protecting the whole application, we can protect only the most error-prone regions. In this way, we can balance the error rate and performance.

4.2. Fault Injection Experiments

We perform fault injection experiments to determine the most error-prone kernels of the target application. We utilize the debugger-based fault injection tool (Öz and Karadaş (2022)), which creates various points to inject fault for each kernel function based on provided information during the profiling phase. The tool consists of 5 distinct phases, as shown in Figure 4.2. In Phase 0, *configuration setup*, the user sets the parameters such as application name and fault injection information. In Phase 1, *profiling*, the target application is executed through the debugger (cuda-gdb) and collects information such as the number of blocks and threads the application uses and the golden output. In Phase 2, *fault map generation*, the tool generates a fault distribution map to determine the fault locations and timing based on information provided in the previous phases. The fault type

considered in this study is the corruption of data stored in the register file, i.e., a bit flip in the target register. In Phase 3, *fault injection*, faults are injected during the execution of the specified instruction. A breakpoint is set while the application is executed through the debugger. The execution is paused at the specified instruction execution, and the value of the target register at the time is corrupted by flipping the target bit. After the injection, the application continues. In Phase 4, the tool provides the user with the fault injection results and the SDC cases' resulting output. We use SDC rates to quantify the kernel functions' soft error vulnerability in our target CUDA programs.

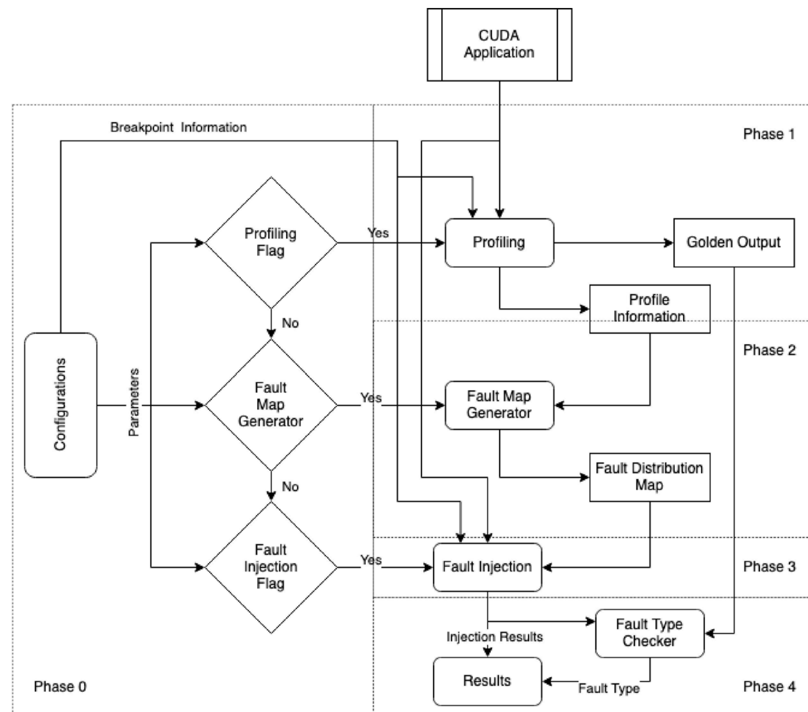


Figure 4.2. Diagram of the Regional Fault Injection Tool (Source: Öz and Karadaş (2022)).

4.3. Custom Compiler

We build our custom compiler on top of LLVM (Lattner and Adve (2004)). As shown in Figure 4.1, our framework has four major parts: 1) Code Annotation, 2) Output Replication, 3) Redundant Kernel Execution, and 4) Output Comparison.

4.3.1. Code Annotation

Code annotations, also known as directives, are usually used to mark a piece of code to ensure the compiler process the code differently. We create a custom directive based on *pragma* to mark will-to-be-replicated function calls. By using code annotation, we enable selective redundancy. The syntax of our custom directive is as follows:

#pragma redundant in <input> *out* <output> *scheme* <Scheme>

The programmer must insert the given pragma just after the kernel execution call in the target application by specifying the input, output, and redundant scheme by *in*, *out*, and *scheme* clauses, respectively. The *in* and *out* clauses require the variables defined earlier in the application. Even though *input* is required, it is not in the sphere of replication. Currently, it is a placeholder for future works. The scheme clause has twelve valid options, as seen in Figure 4.3. The valid schemes are listed as follows: SMKE (Small Memory Multiple Kernel Execution), SMKES (Small Memory Multiple Kernel Execution with Streams), SXTSKE (Small Memory X dimension Thread based Single Kernel Execution), SYTSKE (Small Memory Y dimension Thread based Single Kernel Execution), SXBSKE (Small Memory X dimension Block based Single Kernel Execution), SYBSKE (Small Memory Y dimension Block based Single Kernel Execution); LMKE (Large Memory Multiple Kernel Execution), LMKES (Large Memory Multiple Kernel Execution with Streams), LXTSKE (Large Memory X dimension Thread based Single Kernel Execution), LYTSKE (Large Memory Y dimension Thread based Single Kernel Execution), LXBSKE (Large Memory X dimension Block based Single Kernel Execution), LYBSKE (Large Memory Y dimension Block based Single Kernel Execution). We explain those schemes in Section 4.3.3.1 and Section 4.3.3.2.

Our compiler generates the external function calls and instructions required for our approaches. However, in this part, we provide code snippets to express the main idea behind our implementation. The code snippet given in Listing 4.1 presents the initial code, where the programmer has marked *fdtd_step2_kernel* kernel function in *Fdtd2d* application for redundant execution. We provide the related code snippets in the following

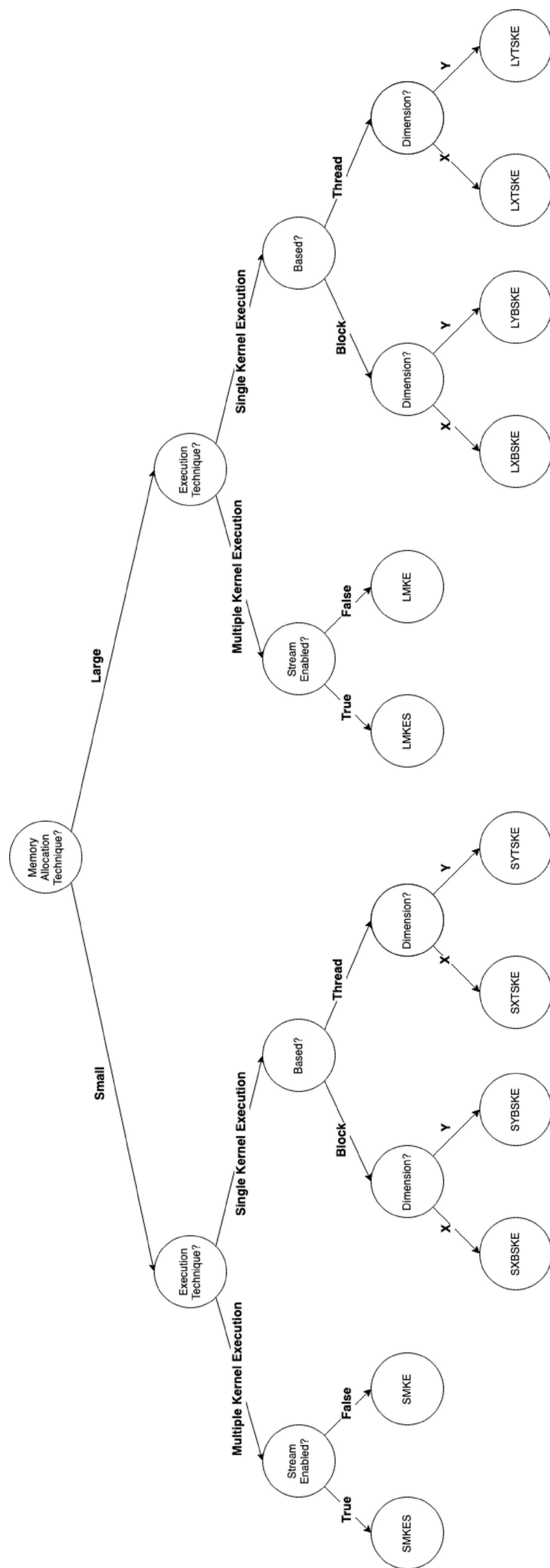


Figure 4.3. Valid Redundancy Schemes.

sub-sections to demonstrate how our approaches generate necessary code by modifying the given code.

```
1
2  __global__ void fdtd_step2_kernel(int nx, int ny, float *ey, float *hz, int t,  float *ex)
3  {
4          int j = blockIdx.x * blockDim.x + threadIdx.x;
5          int i = blockIdx.y * blockDim.y + threadIdx.y;
6          if ((i < _PB_NX) && (j < _PB_NY) && (j > 0))
7          {
8                  ex[i * NY + j] = ex[i * NY + j] - 0.5f*(hz[i * NY + j] - hz[i * NY + (j-1)]);
9          }
10 }
11 ...
12 int main(){
13     ...
14         dim3 block(DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y);
15         dim3 grid (NY/DIM_THREAD_BLOCK_X , NX/DIM_THREAD_BLOCK_Y);
16         fdtd_step2_kernel<<<grid,block>>>(nx, ny, ey_gpu, hz_gpu, t, ex_gpu);
17         #pragma redundant in ny out ex_gpu scheme LMKE
18     ...
19 }
```

Listing 4.1 Annotated Initial *Fdtd2d* Code.

As shown in Listing 4.1, the function call *fdtd_step2_kernel* is annotated. The annotation indicates that variable ny is the *input*; variable ex_gpu is the *output*; selected *redundant scheme* is the *LMKE*. The provided information passes into the LLVM IR level as metadata by Clang. The generated LLVM IR code is given in Listing 4.2. It can be seen that function call *fdtd_step2_kernel* in line 16 has metadata, and the metadata that can be seen in line 20, has the provided information.

```
1      ...
2          %call18 = call i32 @cudaConfigureCall(i64 %52, i32 %54, i64 %58,
3              i32 %60, i64 0, %struct.CUstream_st* null)
4          %tobool19 = icmp ne i32 %call18, 0
5          br i1 %tobool19, label %kcall.end21, label %kcall.configok20
6
7      kcall.configok20:
8       ; preds = %kcall.end
9          %61 = load i32, i32* %nx.addr, align 4
10         %62 = load i32, i32* %ny.addr, align 4
11         %63 = load float*, float** %ey_gpu, align 8
12         %64 = load float*, float** %hz_gpu, align 8
13         %65 = load i32, i32* %t, align 4
14         %66 = load float*, float** %ex_gpu, align 8
15         call void @_Z17fdtd_step2_kerneliiPfS_iS_(i32 %61, i32 %62, float* %63,
16             float* %64, i32 %65, float* %66), !Redundancy !3
17         br label %kcall.end21
18
19     ...
20 !2 = !{!"Inputs_&ny_Outputs_&ey_gpu_Scheme_&MKE"}
```

Listing 4.2 Generated LLVM IR Code.

4.3.2. Output Replication

Our framework detects the output-to-be-replicated by using the metadata that we create using our custom pragma directive. Our *output replication* strategy consists of three steps as follows:

1. *Copy declaration:* We declare a new copy of the output-to-be-replicated by considering the original output type.

2. *Memory allocation:* We allocate memory space in the GPU using the *cudaMalloc* function, which is the default function to allocate memory in the GPU.

3. *Initialization:* Since we need to initialize the output variables due to the utilization of the initial values in the target function executions, we copy the data values from the original output variable into the redundant copy. Since we have the initialized values in the GPU memory, we utilize the device-to-device memory copy operation (via *cudaMemcpyDevicetoDevice* parameter) to avoid the overhead caused by copying from the host to the device.

We classify our schemes into two subcategories by their memory allocation technique: large memory and small memory schemes. We give detail about this distinction in the following sections. If we employ large memory schemes, we generate two redundant copies. However, for small memory schemes, we firstly create a checkpoint of the output-to-be-replicated by following the three steps listed below:

1. *Variable declaration:* We declare a new copy of the output-to-be-replicated by considering the original output type.

2. *Memory allocation:* We allocate memory space in the CPU by using *malloc* function.

3. *Initialization:* We use *cudaMemcpy* function, which allow us to transfer data from device to host and vice versa, with the option *cudaMemcpyDevicetoHost*. Hence, we create a new variable containing the output-to-be-replicated's initial data.

After creating the checkpoint, we create only one redundant output, perform two computations, then check if there is an error by comparing the output. If an error occurs, we restore the original output with the checkpoint. Finally, perform another computation and determine the correct output. Therefore, we can reduce the memory requirement by increasing memory copy operation.

After performing redundant executions, we compare the outputs to determine if an error occurs and correct the output in case of erroneous computation. We implement detection and majority voting functions, where detection compares two outputs and detects mismatch if there is an error, and majority voting performs both detection and correction by considering the majority of three outputs. We generate codes necessary for the functions as CUDA kernels on the device side and their required initialization codes as functions on the host side during compilation.

4.3.3. Redundant Kernel Execution

As the foundation of our redundancy framework, we generate the necessary code to execute our target kernel function computations redundantly in this part. As mentioned, we create twelve distinct redundancy schemes to overcome memory limitations and exploit the massively parallel architecture of GPUs.

As mentioned in Section 5.2, our experiments show that kernel executions are the leading cause of the overhead. To reduce overhead, we design and implement redundancy schemes by considering their *execution techniques* and *memory allocation techniques*.

4.3.3.1. Memory Allocation Techniques

Our redundancy-based techniques tend to increase the memory usage of the target applications. As with many components, GPUs might have memory limitations. Therefore, we offer two distinct memory allocation techniques to reduce memory dependence: large memory and small memory.

Our large memory-based techniques use the majority voting function to compare

redundant outputs. Since the majority voting function requires three redundant outputs, we create three redundant outputs which might not fit in the memory simultaneously, and it might cause incorrect execution or execution failure. Besides that, three redundant outputs need to be computed. Therefore, it increases the overhead.

Even though our large memory-based techniques are suitable for many applications and GPUs, we offer small memory-based techniques to protect memory-intensive applications and GPUs with memory limitations.

There are two main motivations behind our small memory-based techniques:

1. Large memory techniques require three redundant copies. Therefore, those three redundant copies must be simultaneously in the global memory. Even though this situation does not affect applications with small memory configurations, it may affect applications with large configurations.

2. The execution might be error-free. Therefore, it is not required to execute the third kernel function. It would reduce the overhead.

4.3.3.1.1 Large Memory

Figure 4.4 shows the general form of the large memory-based techniques. Firstly, we declare the *second copy* and *third copy* with the same type as original output. Then, we allocate memory in the GPU using the *cudaMalloc* function, and we equalize the copies by using the *cudaMemcpy* function with *cudaMemcpyDeviceToDevice* option in case the output has initial data. Secondly, we execute the redundant kernel based on the selected approach. Eventually, it produces three outputs regardless of execution techniques. Lastly, we use the majority voting function to detect and correct the erroneous values. In the code snippet given in Listing 4.3 and follows the large memory technique, ex_gpu, ex_gpu1, ex_gpu2 and fdtd_step2_kernel correspond to output, copy_1, copy_2 and kernel in Figure 4.4.

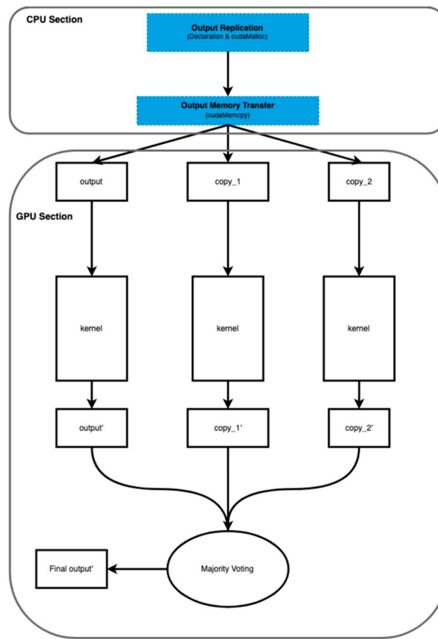


Figure 4.4. General Form of Large Memory Based Techniques.

```
1
2  int main(){
3
4      ...
5      float *ex_gpu1;
6      float *ex_gpu2;
7      cudaMalloc((void **)&ex_gpu1, sizeof(float * N);
8      cudaMalloc((void **)&ex_gpu2, sizeof(float) * N);
9
10     cudaMemcpy(ex_gpu1, ex_gpu, sizeof(DATA_TYPE) * N, cudaMemcpyDeviceToDevice);
11
12     cudaMemcpy(ex_gpu2, ex_gpu, sizeof(DATA_TYPE) * N, cudaMemcpyDeviceToDevice);
13
14    fdtd_step2_kernel<<<grid,block>>>(nx, ny, ey_gpu, hz_gpu, t, ex_gpu);
15
16    fdtd_step2_kernel<<<grid,block>>>(nx, ny, ey_gpu, hz_gpu, t, ex_gpu1);
17
18    fdtd_step2_kernel<<<grid,block>>>(nx, ny, ey_gpu, hz_gpu, t, ex_gpu2);
19
20     majorityVoting<<< grid, block >>>(ex_gpu, ex_gpu_1, ex_gpu2, N);
21
22     ...
23
24 }
```

Listing 4.3 CUDA Code for Large Memory Based Technique.

4.3.3.1.2 Small Memory

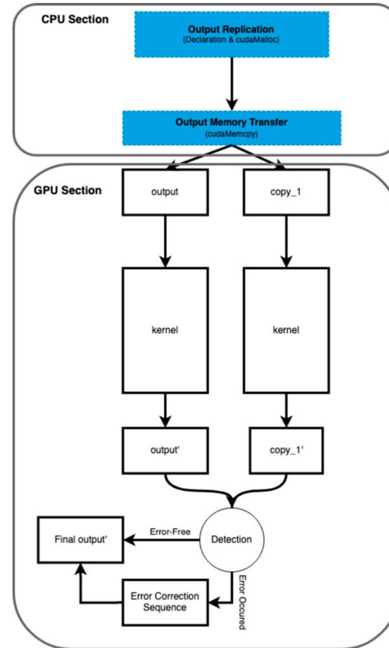


Figure 4.5. General Form of Small Memory-Based Techniques.

Figure 4.5 shows the general form of the small memory-based techniques. First, we create two variables, one for the host and one for the device. We use them as flags to detect if there is an error. Then we create the checkpoint to store the initial data of output in the host. After that, we create the *second copy* and equalize it with the original output. Then we execute the kernel. After executing the kernel twice, we execute the detection function, which will detect if there is an error. If there is no error, execution continues normally. Otherwise, it starts *error correction* sequence.

The *error correction* mechanism assumes that two of the three computed outputs are equal to each other. Suppose the original output and second copy are not equal to each other, therefore, the third output.

The *error correction* mechanism assumes that two of the three computed outputs are equal to each other. Suppose the original output and second copy are not equal to each other. The third one is equal to one of them. Therefore, without any further detection, we can restore the original output from the checkpoint and execute the given kernel one more

time with the original output to produce the correct output.

4.3.3.2. Execution Techniques

Redundant multithreading increases the execution time of the target applications. Like many components, GPUs come with different configurations, such as the number of cores and clock rate. Therefore, we offer two distinct execution techniques to exploit GPU features: multiple kernel execution and single kernel execution.

Our multiple kernel execution based techniques enable redundant execution with multiple kernel launches in series or streams. Even though those techniques are adequate, they might not exploit the massively parallel architecture of GPUs. Therefore, we offer single kernel execution based techniques. Those techniques alter the will-to-be-replicated function's signature and its launch configurations to perform redundancy.

4.3.3.2.1 Multiple Kernel Execution (MKE)

In those schemes, we replicate the function calls of the annotated functions by providing redundant outputs. Essentially, we launch redundant kernel functions back-to-back in this approach. For instance, for the annotated code given in Listing 4.1, we create additional function calls of the annotated function by providing created outputs as the output arguments. Therefore, redundant function calls write to redundant outputs. Listing 4.4 presents the output replication (*ey_gpu1* and *ey_gpu2* in lines 2-3), redundant kernel function calls (lines 13 and 15), and majority function call with redundant output arguments (line 17).

The function calls in *MKE* schemes execute sequentially, and CUDA supports parallelization among kernels (via CUDA Streams). We do not exploit the feature of the GPUs with this scheme. Therefore, we offer the following scheme.

```
1  int main(){
2
3      ...
4
5      float *ey_gpu1;
6      float *ey_gpu2;
7      cudaMalloc((void **)&ey_gpu1, sizeof(float * N);
8      cudaMalloc((void **)&ey_gpu2, sizeof(float) * N);
9
10     cudaMemcpy(ey_gpu1, ey_gpu, sizeof(DATA_TYPE) * N, cudaMemcpyDeviceToDevice);
11
12     cudaMemcpy(ey_gpu2, ey_gpu, sizeof(DATA_TYPE) * N, cudaMemcpyDeviceToDevice);
13
14    fdtd_step1_kernel<<<grid,block>>>(nx, ny, _fict_gpu, ex_gpu, hz_gpu, t, ey_gpu);
15
16    fdtd_step1_kernel<<<grid,block>>>(nx, ny, _fict_gpu, ex_gpu, hz_gpu, t, ey_gpu1);
17
18    fdtd_step1_kernel<<<grid,block>>>(nx, ny, _fict_gpu, ex_gpu, hz_gpu, t, ey_gpu2);
19
20    majorityVoting<<< grid, block >>>(ey_gpu,ey_gpu_1, ey_gpu2, N);
21
22 }
```

Listing 4.4 CUDA Code for LMKE Scheme.

4.3.3.2.2 Multiple Kernel Executions with Stream Enabled (MKES)

A *stream* in CUDA is defined as a sequence of kernel executions issued by the host code (Harris (2022)). The primary motivation behind CUDA streams is to make the independent kernel executions parallel.

```
1
2  int main(){
3      /*
4          Memory Allocation and Memory Copy Operations
5      */
6      cudaStream_t stream[3];
7      cudaStreamCreateWithFlags(&stream[0],1);
8      fdtd_step1_kernel<<<grid,block,0, stream[0] >>>(nx, ny, _fict_gpu, ex_gpu, hz_gpu, t, ey_g   pu);
9      cudaStreamCreateWithFlags(&stream[1],1);
10     fdtd_step1_kernel<<<grid,block,0, stream[1] >>>(nx, ny, _fict_gpu, ex_gpu, hz_gpu, t, ey_gpu1);
11     cudaStreamCreateWithFlags(&stream[2],1);
12     fdtd_step1_kernel<<<grid,block,0, stream[2] >>>(nx, ny, _fict_gpu, ex_gpu, hz_gpu, t, ey_gpu2);
13
14     majorityVoting<<< grid, block >>>(ey_gpu,ey_gpu_1, ey_gpu2, N);
15
16 }
```

Listing 4.5 CUDA Code for LMKES Scheme.

Since redundant functions in our *MKE* schemes are executed sequentially and do not conflict with their variable usage, we can enable CUDA streams to reduce kernel ex-

ecution. By using streams in the *MKES* scheme, we target to reduce the execution time by overlapping kernel executions. *MKES* issues redundant kernel executions in different non-default CUDA streams and potentially executes the redundant computations in parallel. Therefore, we can exploit the massively parallel architecture of GPUs. There are two main alterations needed for the implementation of *MKES*:

1. Creating streams using *cudaStreamCreateWithFlags* function (Lines 7, 9, and 11 in Listing 4.5)

2. Modifying the kernel call to give the created stream as an argument (Lines 8, 10, and 12 in Listing 4.5).

4.3.3.2.3 Single Kernel Execution (SKE)

Even though the multiple kernel executions are practical and effective redundancy approaches, the function launches could be expensive (Lingqi Zhang and Matsuoka (2019)), and *MKE* does not fully exploit the massively parallel architecture of GPUs. By this motivation, we offer another technique: single kernel execution with redundant threads. Our *SKE* scheme performs redundancy by either replicating threads in each existing block or replicating blocks with the same number of threads. Additionally, we consider the replications in X or Y dimensions. We refer to those different schemes as thread-based single kernel execution (TSKE), block-based single kernel execution (BSKE), single kernel execution in X (XSKE), and single kernel execution in Y (YSKE). As a result, we have the following four schemes with their basic replication methods:

- **XTSKE:** Intra-block (multiple threads in existing blocks) replication in X dimension.

- **YTSKE:** Intra-block replication in Y dimension.

- **XBSKE:** Inter-block (multiple blocks) replication in X dimension.

- **YBSKE:** Inter-block replication in Y dimension.

Figure 4.6 demonstrates the grid organization for the original kernel execution, which consists of four blocks, each having four threads. There are two blocks in both (X and Y) dimensions of the grid, and each block dimension includes two threads.

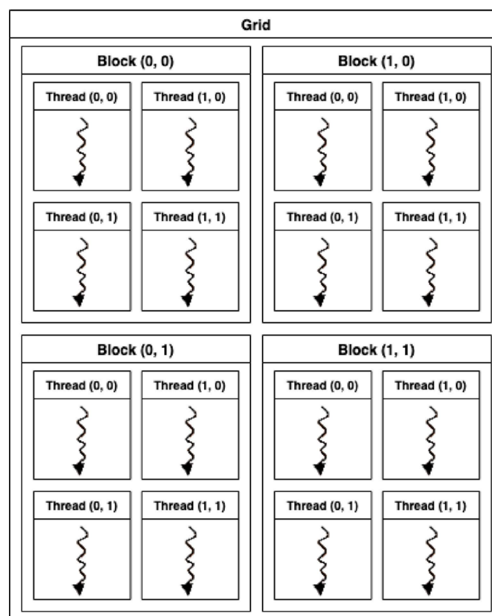
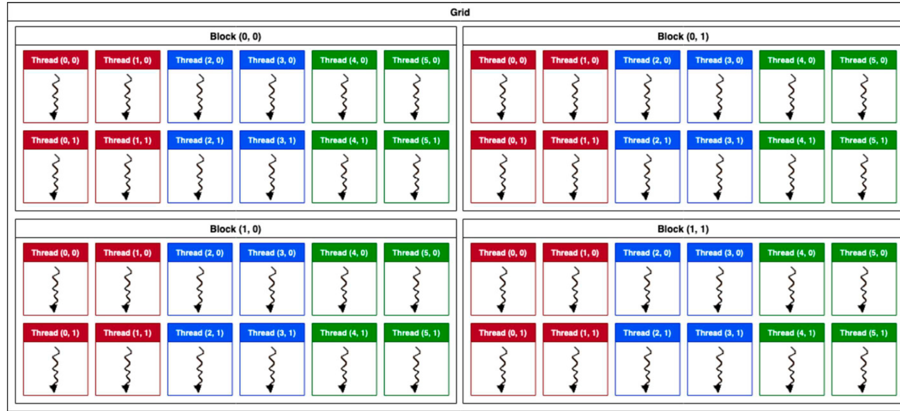


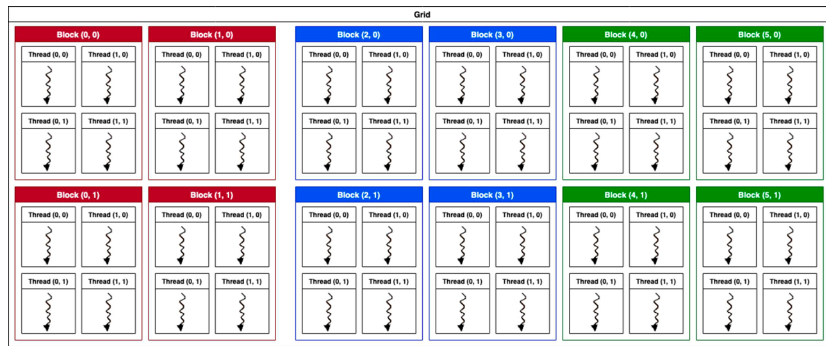
Figure 4.6. Grid Organizations for Original Execution.

Figure 4.7 and Figure 4.8 present an example grid and block organization for the large memory based single kernel execution schemes. It can be observed that the number of blocks and threads is tripled. We color the rest of the figures to distinguish redundant blocks and threads. In the small memory-based schemes, grid and block organizations are similar to large memory-based schemes. Instead of tripling, we duplicated the number of threads and blocks.

Figure 4.7a demonstrate the thread organization of our *Large Memory XTSKE* (LXTSKE) schemes. In the original grid, we have two threads in the X dimension. As seen in the Figure 4.7a, with *LXTSKE* approach, we have six threads in the X dimension while the number of blocks in the grid does not change. The threads colored in red are responsible for the original output, while the threads colored in blue and green are responsible for the redundant *copy_1* and redundant *copy_2*, respectively. On the other hand, with *Small Memory XTSKE* (SXTSKE) approach, we have four threads in the X dimension while the number of blocks in the grid does not change.



(a) LXTSKE Scheme.

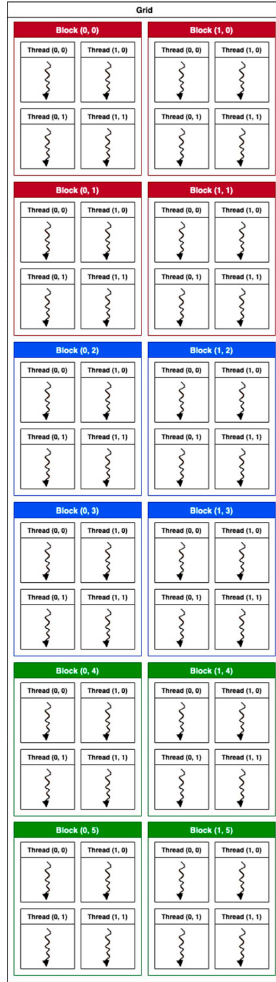


(b) LXBSKE Scheme.

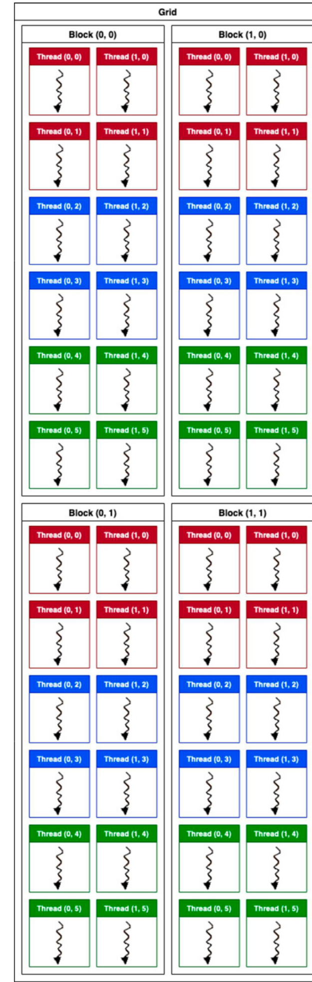
Figure 4.7. Grid Organizations for LX*SKE Schemes.

To enable redundancy with single kernel execution, we launch our kernel function with redundant threads by providing additional outputs used by redundant threads. Moreover, we modify the target kernel codes for thread ID and correct output identifications. Based on those operations for the *SKE* scheme, we have three common alterations in the target code as follows:

1. **Kernel signature modification:** Since redundant threads require to be updated with different output data structures (similar to *MKE* techniques), we update the argument list of our kernel function by including one additional output for small memory techniques and two additional output for large memory techniques. We define extra output variables in our host code like *MKE* schemes and provide them



(a) LYBSKE Scheme



(b) LYTSKE Scheme

Figure 4.8. Grid Organizations for LY*SKE

as arguments while launching the kernel function. Additionally, we include another argument, *original*, to pass the original thread or block size, which enables thread ID and output identification for the redundant computations. For instance, we have the original kernel definition as follows:

__global__ void kernel(..., out)

Our modified kernel definition for large memory single kernel execution scheme is as follows:

__global__ void kernel(..., out, copy_1, copy_2, original)

Table 4.1. ThreadID Modification Codes for Our SKE-based Schemes.

	Original	BSKE		TSKE	
		X	Y	X	Y
X - ThreadID	blockIdx.x * blockDim.x + threadIdx.x	(blockIdx.x % OriginalBased) * blockDim.x + threadIdx.x		blockIdx.x * OriginalBased + (threadIdx.x % OriginalBased)	
Y - ThreadID	blockIdx.y * blockDim.y + threadIdx.y		(blockIdx.y % OriginalBased) * blockDim.y + threadIdx.y		blockIdx.y * OriginalBased + (threadIdx.y % OriginalBased)

This modification is the same for all large memory single kernel execution schemes. Our modified kernel definition for the small memory-based single kernel execution scheme is as follows:

__global__ void kernel(..., out, copy_1, original)

2. **Thread ID modification:** We need to update the *thread ID* computation code for the kernel functions working with global thread IDs to determine the operations required by individual threads. Depending on which redundant copy of the computation is being performed by the current thread, we convert *thread ID* by considering the original number of threads. For instance, for execution with 32 threads in one block, we have 32x3 threads in the same block in our *LTSKE* scheme. In redundant execution, threads with *thread ID*s *0*, *32*, and *64* should perform exactly the same operations by targeting redundant data. While *Thread 0* needs to work with original output, *Thread 32* utilizes first redundant output, and *Thread 64* writes into the second redundant output. In our kernel function, we need to modify the *thread ID*s accordingly. Since we replicate threads differently in our different *SKE* schemes, *thread ID* modification also differs. Table 4.1 presents the required *thread ID* modifications for *SKE* schemes, and if no modification is required, the cell is left blank.

3. **Output identification:** Based on thread or block numbers (*threadIdx* or *blockIdx* CUDA variables), we redirect output values to the target redundant output variable. The skeleton of the code inserted at the beginning of the kernel function is given in Listing 4.6. Firstly, we define a local variable as a pointer to the relevant output variable (metaOutput in the given code) in the redundant threads. Additionally, we have a control statement to check the ID and decide the output variable to be used in the kernel execution.

```
1      float* metaOutput;
2      if(redundantId / original == 0){
3          metaOutput = out;
4      }
5      else if(redundantId / original == 1){
6          metaOutput = copy_1;
7      }
8      else if(redundantId / original == 2){
9          metaOutput = copy_2;
10     }
```

Listing 4.6 Output Identification Code

Since redundant threads correspond to the different original thread IDs, we perform the modification by considering the *SKE* scheme. *redundantID* in Listing 4.6 refers to blockId.x, blockId.y, threadId.x and threadId.y for *XBSKE*, *YBSKE*, *XTSKE*, and *YTSKE*, respectively.

4.3.4. Output Comparison

We have two functions that we use either one to compare the redundant outputs: detection and majority voting.

Our *detection* function takes two outputs, the output size, and an error variable, as arguments. The detection function compares each element (based on the assumption that output is a data structure consisting of multiple elements) of two outputs, and if any two values are not equal, it sets the error variable to one. The code snippet given in Listing 4.7 presents the inspired code for our detection function.

```
1
2  __global__ void detection(float* Output, float* SecondCopy,
3      int Size, int* Error ){
4   int ThreadIdX = blockIdx.x * blockDim.x + threadIdx.x;
5   int ThreadIdY = blockIdx.y * blockDim.y + threadIdx.y;
6   int ThreadId = blockDim.x * ThreadIdY + ThreadIdX;
7
8   if (ThreadId < Size &&
9      Output[ThreadId] != SecondCopy[ThreadId])
10   Errors[0] = 1;
11
12 }
```

Listing 4.7 Inspired Detection Function.

Our majority voting function takes three outputs and the output size as arguments and assumes that at least two outputs are equal. Therefore, we can say that if the *second*

copy and the *third copy* are equal to each other, we can write the *third copy's* value to the *first copy* without checking. If they are not equal, the *second copy's* or the *third copy's* value is equal to *first copy's* value. Hence, we do not have to re-write the correct value. The code snippet given in Listing 4.8 presents the inspired code for our majority voting function.

```
1
2  __global__ void majorityVoting(float* Output, float* SecondCopy,
3      float* ThirdCopy, int Size){
4   int ThreadIdX = blockIdx.x * blockDim.x + threadIdx.x;
5   int ThreadIdY = blockIdx.y * blockDim.y + threadIdx.y;
6   int ThreadId = blockDim.x * ThreadIdY + ThreadIdX;
7
8   if (ThreadId < Size &&
9      SecondCopy[ThreadId] == ThirdCopy[ThreadId])
10   Output[ThreadId] = ThirdCopy[ThreadId];
11 }
```

Listing 4.8 Inspired Majority Voting Function.

CHAPTER 5

EXPERIMENTAL RESULTS

In this chapter, we present the experiments. Firstly, we mention our setups, and then we give our results. Our results show the relationships between redundancy executions, performance, and error rates.

5.1. Experimental Setup

For evaluating our redundancy approaches, we utilize 19 kernel functions from 8 different applications from the PolyBench benchmark suite (Grauer-Gray et al. (2012)). Specifically, we consider all kernel functions from the following applications: *2DConvolution*, *3DConvolution*, *3mm*, *Correlation*, *Covariance*, *Fdtd2d*, *Gemm*, and *Gramschmidt*.

For our compiler-level redundancy approaches, we use Clang 10.0.1 to compile and LLVM 10.0.1 to generate additional code. We customize the Clang with custom *pragma* to annotate the will-to-be-replicate kernels. We use Ninja as a build system to build our custom LLVM and Clang (Martin (2012)). Our CUDA version is 10.0.

We utilize two GPU devices with different compute capabilities and the number of available parallel execution units for our evaluation platform. Specifically, we use Tesla K80 provided by the COLAB environment (Colaboratory-Frequently Asked Questions (2022)) and Quadro P620 GPU in our local infrastructure. Table 5.1 gives the main features of those GPU devices.

We build our compilation framework on LLVM Framework version 10.0.1 and generate the target executables for redundant execution schemes using our custom Clang compiler. We use *nvprof*, which NVIDIA provides as a built-in tool for the CUDA programs to collect the execution times for the kernel functions and the specific operations such as memory copy, and kernel function execution (Profiler User's Guide (2022)).

Table 5.1. Salient Characteristics of GPU Devices Used in Our Experiments.

	Tesla K80	**Quadro P620**
CUDA Compute Capability	3.7	6.1
Global memory size	11441 MB	4042 MB
Multiprocessors	13 MP	4 MP
CUDA Cores per Processor	192 CUDA Cores	128 CUDA Cores
GPU Max Clock rate	824 MHz (0.82 GHz)	1443 MHz (1.44 GHz)
Memory Clock rate	2505 Mhz	3004 Mhz
Memory Bus Width	384-bit	128-bit
L2 Cache Size	1572864 bytes	524288 bytes
Total number of registers available per block	65536	65536
Maximum number of threads per multiprocessor	2048	2048
Maximum number of threads per block	1024	1024
Max dimension size of a thread block (x,y,z)	(1024, 1024, 64)	(1024, 1024, 64)
Max dimension size of a grid size(x,y,z)	(2147483647, 65535, 65535)	(2147483647, 65535, 65535)

5.2. Experimental Results

For our partial redundancy evaluation, we perform fault injection experiments to determine the most vulnerable kernel function(s) in the target application by utilizing a debugger-based fault injection tool (Öz and Karadaş (2022)). In different setups, we mark a different kernel function in our programs for fault injection target to evaluate the soft error vulnerability of each kernel function. While the tool yields *masked, crash,* and *SDC rates*, we only use the SDC rates as the soft error vulnerability metric of the kernel function.

We present the SDC rates of each kernel function we obtained from our fault injection experiments in Figure 5.1. In addition, we present the execution times of the corresponding kernels in Table 5.2. In this part of our experiments, we use five distinct applications.

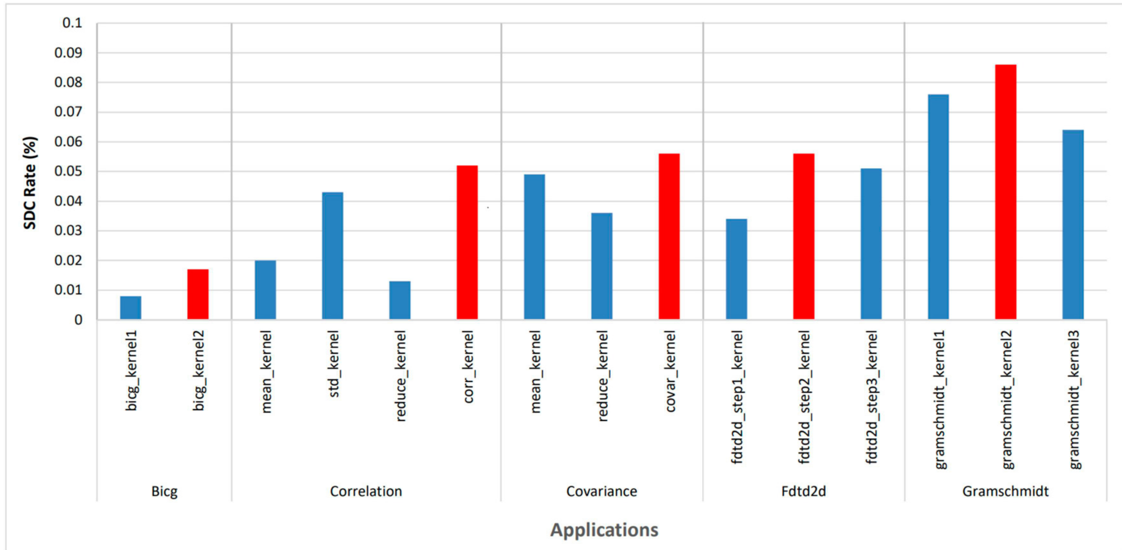


Figure 5.1. SDC Rates for the Kernel Functions.

We mark the most error-prone kernels with the highest SDC rate for each application with red in Figure 5.1. For this part of our experimental study, we use our *LMKE* scheme to perform redundant executions with different redundancy scenarios on Quadro P620 GPU. Precisely, we execute our target applications with the following scenarios: 1) No Redundancy, where we run the program without any altering, 2) Full Redundancy, where we execute all the kernel functions in the application redundantly, that is, three times, 3) One kernel redundant, where we execute the most error-prone kernel in the application.

Figure 5.2 indicates the normalized execution times of the target application with different redundancy scenarios. For each application in the figure, we mark the execution scenario in which the kernel function with the highest SDC rate is replicated, e.g. *fdtd2d_step2_kernel* in *Fdtd2d*, *covar_kernel* in *Covariance*.

If we focus on the *Bicg*, we can see the redundant execution scenario, where the *bicg_kernel2* is with the highest SDC rate among *Bicg* is replicated and lasts in a significantly shorter time than the full redundant case. Our selective scenario yields performance gain in comparison to the full redundant scenario. However, *Correlation* and *Covariance* do not act in the similar way. Even though *corr_kernel* of the *Correlation* is the most error-prone kernel function and as it can be observed that the execution time

Table 5.2. Execution times for the kernel functions.

Application Name	Kernel Name	Execution Time (s)
Correlation	mean_kernel	0.003
	std_kernel	0.003
	reduce_kernel	0.004
	corr_kernel	5.945
Covariance	mean_kernel	0.003
	reduce_kernel	0.001
	covar_kernel	6.174
Bicg	bicg_kernel1	8.056 x 10^{-3}
	bicg_kernel2	22.496 x 10^{-3}
Fdtd2d	fdtd2d_step1_kernel	8.056 x 10^{-3}
	fdtd2d_step2_kernel	2.703 x 10^{-3}
	fdtd2d_step3_kernel	3.043 x 10^{-3}
Gramschmidt	gramschmidt_kernel1	0.922 x 10^{-3}
	gramschmidt_kernel2	0.008 x10^{-3}
	gramschmidt_kernel3	4.344 x 10^{-3}

of *corr_kernel* takes almost six hundred times longer than the other kernels combined. Therefore, it dominates the other kernels. Hence, the difference between the full redundancy and only *corr_kernel* redundancy is insignificant, as observed in Figure 5.2. On the other hand, the SDC rate of the *std_kernel* is closer to the SDC rate of *corr_kernel* as seen in Figure 5.1. Since the execution of the *std_kernel* lasts significantly shorter than the execution time of the *corr_kernel* as in Table 5.2, it would be advantageous if we replicate both of the kernels to decrease the SDC rate without significant performance loss. Sim-

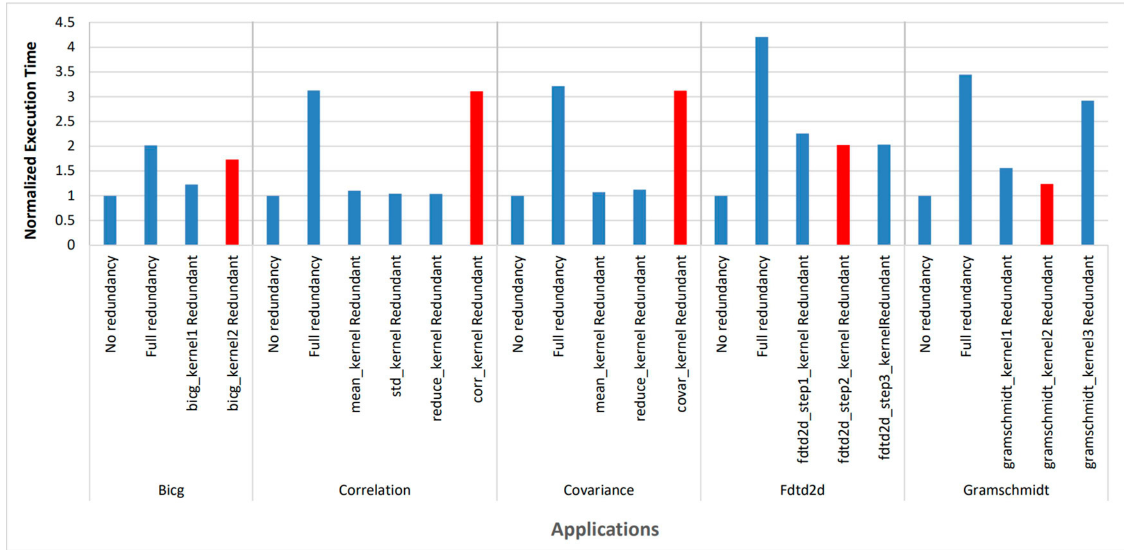


Figure 5.2. Normalized Execution Times for the Redundant Executions.

ilarly, the *covar_kernel* of the *Covariance* dominates the application since the execution times of the rest of the kernels are negligible compared to *covar_kernel*. On the other hand, the SDC rate of the *mean_kernel* is closer to the SDC rate of *covar_kernel* as seen in Figure 5.1. If we replicate both kernels, we achieve potentially lower SDC rates with a negligible execution time difference. We must note that the time difference between only *covar_kernel* redundancy scenario and the full redundancy scenario is negligible. Hence, the best redundancy option is full redundancy. The observed trends in *Correlation* and *Covariance* indicate that for applications that are dominated by a single kernel function in terms of the execution time, applying for full redundancy instead of selective redundancy can be a good option.

Unlike the other applications, the proportion between the execution time and the SDC rate is inverse in *Fdtd2d* and *Gramschmidt*. For instance, while the execution time of *fdtd2d_step2_kernel* is the shortest among the other kernel functions in the application as shown in Table 5.2, its SDC rate is the highest as shown in Figure 5.1. The scenario with the only *fdtd2d_step2_kernel* redundant has the lowest among all the redundancy scenarios as shown in Figure 5.2. The kernel has the highest SDC rate. Therefore, we get a sufficient SDC rate improvement by sacrificing a minor execution time. On the other hand, the SDC rate of the *fdtd2d_step3_kernel* is close to the *fdtd2d_step2_kernel*,

and the difference between their execution times is not significant as seen in Table 5.2. Hence it would be advantageous if we replicate both of the kernels. *Gramschmidt* behaves similarly. For example, the *gramschmidt_kernel2* has the highest SDC rate, yet it has the lowest execution time among the other kernels of the application. Therefore replicating only this function or two functions, namely the *gramschmidt_kernel1* and the *gramschmidt_kernel2*, with similar execution times can be advantageous in terms of both performance and reliability.

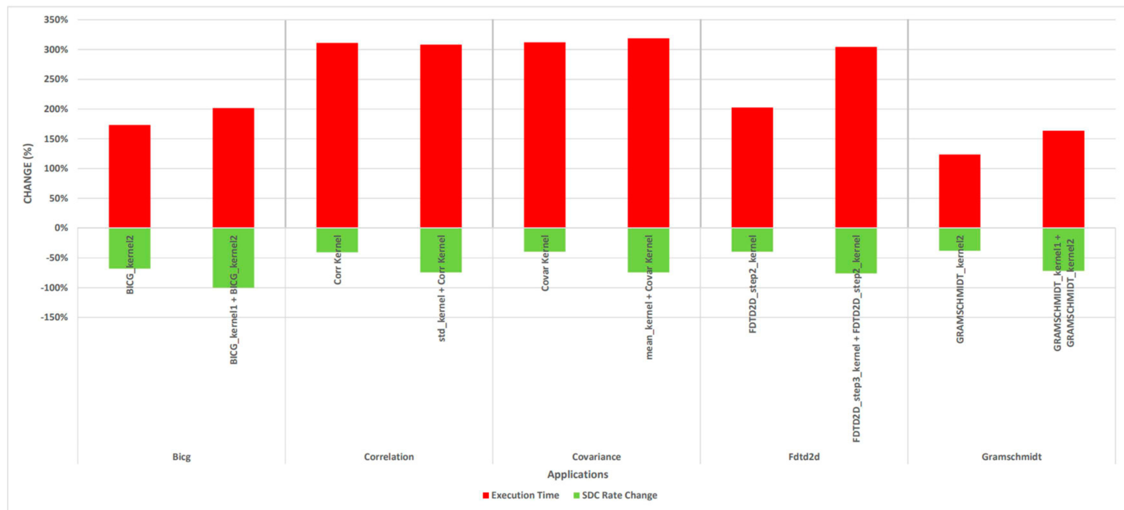
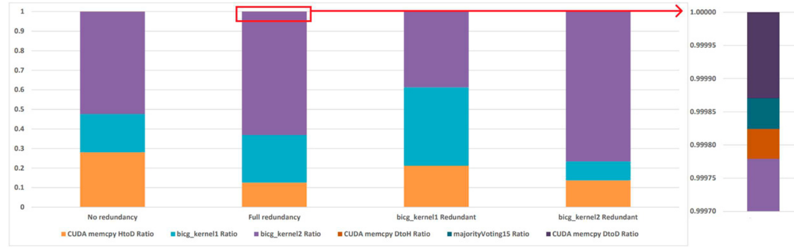
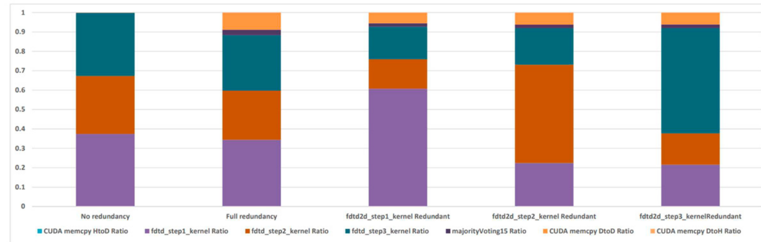


Figure 5.3. The Change in Percentage of SDC Rates and Execution Times for the Redundant Executions.

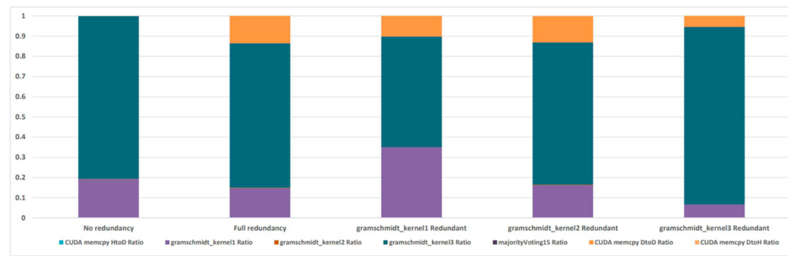
We present the change in percentage for both SDC rates as the vulnerability metric and the execution times in Figure 5.3 to evaluate the performance-reliability balance. We assume that our triple replication approach offers full protection, and our redundantly executed code will run error-free. We compute the SDC rates based on this. We construct two redundant execution scenarios for each application: replicating only the most vulnerable kernel function and replicating the most vulnerable two kernel functions. For *Bicg*, since it consists of two kernel functions, we demonstrate the full redundancy and the most vulnerable function redundancy scenario. In the full redundancy scenario, we replicate two kernel functions, which lead to SDC rate drops to zero (100% decrease). On the other hand, the reliability gain obtained from the replication of the *bicg_kernel2* is limited ($\sim$ 65%). Therefore, applying for full redundancy would be more useful if we



(a) Execution Time Profile of Bicg



(b) Execution Time Profile of Fdtd2d



(c) Execution Time Profile of Gramschmidt

Figure 5.4. Execution Time Profile of Each Function.

have no time limitation or no tolerance for SDCs. Otherwise, using only *bicg_kernel2* redundancy would be advantageous in terms of performance. As mentioned earlier, the execution time differences are negligible for both *Correlation* and *Covariance*, while the vulnerability gain increases for the two-function replication case. Even though, in Figure 5.3, there is a large percentage difference between only *fdtd2d_step2_kernel* redundant option and both *fdtd2d_step2_kernel* and *fdtd2d_step3_kernel* redundant option, the absolute difference is not significant since the execution times of the kernels are short (see Table 5.2). The performance gain from replicating only the most vulnerable function becomes significant for cases where *Fdtd2d* is executed with a huge amount of data, and the execution takes longer. For *Gramschmidt*, we can see the balance between the vulnera-

bility and the performance for the alternative redundancy scenario. Replicating only the most vulnerable function, i.e., *gramschmidt_kernel2*, provides almost the same ($\sim 30\%$) performance and vulnerability gains. Our selective redundancy evaluation demonstrates that applications' behaviors vary based on their requirements, including the execution time and reliability. As a result, we need to apply the optimal scenario for reliability and performance based on requirements.

Figure 5.4 demonstrates the execution time profile of each function for the redundancy scenarios. For each redundant execution scenario, we measure the percentage of the operations performed during the execution. Especially, we focus on the kernel function executions, the majority function, and the memory copy operations, including the copy of the input from CPU to GPU (*CUDA memcpy HtoD*), the copy of the output from GPU to CPU (*CUDA memcpy DtoH*), and the redundant output copy operations form GPU to GPU (*CUDA memcpy DtoD*). We observe that most of the execution time is spent during the kernel function executions. While the redundant output copy operations also take important time in *Fdtd2d* as shown in Figure 5.4b and *Gramschmidt* as shown in Figure 5.4c, the percentage of those operations is small for the other programs. We provide the small percentages in a more detailed view for Bicg (see Figure 5.4a), however, we remove the details for *Correlation* and *Covariance*, which both spend almost all the time in the dominant kernel function executions, *corr_kernel* and *covar_kernel*, respectively.

Although we include additional memory operations, majority voting functions, and redundant kernel functions in our redundant scenarios, the main reason for the latency in the execution time is the redundant function executions. The majority voting function and the memory operations take negligible time compared to the kernel executions. Hence, we need to focus on decreasing the time spent executing the redundant kernels. It is possible to use the parallel execution units of the GPU by executing the redundant copies in parallel, either using the streams or increasing the number of threads working on the redundant copies.

We develop the other execution techniques to reduce kernel execution's overhead by utilizing the features of the GPUs and their massively parallel architecture. We extend our experiments with other applications from the PolyBench benchmark suite and utilize Tesla K80 GPU provided by the COLAB environment, which contains more parallel

Table 5.3. Configurations of 2D Convolution.

Application	Kernel	Configuration	Grid		Block		Output Size
			X	*Y*	*X*	*Y*	
2D Convolution	convolution2D_kernel	Standard	128	512	32	8	2^{24}
		Large	256	1024	32	8	2^{26}

resources as given in Table 5.1. Even though the additional applications have different characteristics, they mainly have a single kernel, making them incompetent for partial redundancy. Therefore, we perform full redundancy for all the applications to provide consistency.

Firstly, we annotate all kernel functions in target programs with our custom *pragma*. Then we compile the programs with our custom Clang using our custom passes. Finally, we run our executables in our target architectures.

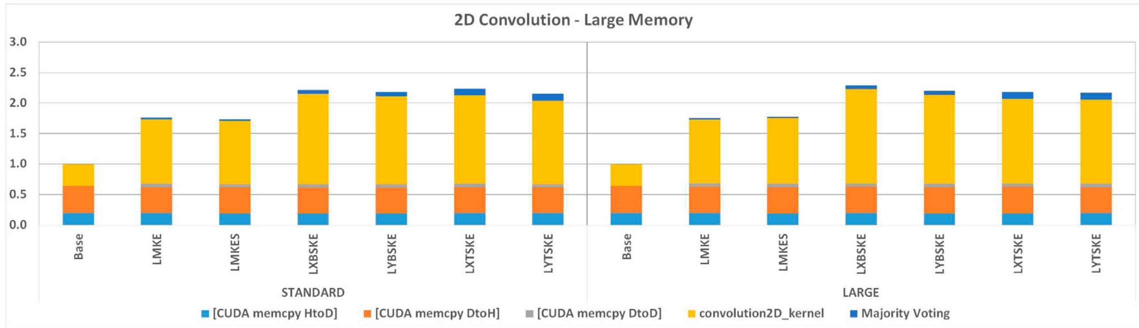
We have twelve different redundant execution cases for each target program. Our redundant execution schemes are listed as follows: SMKE, SMKES, SXTSKE, SYTSKE, SXBSKE, SYBSKE; LMKE, LMKES, LXTSKE, LYTSKE, LXBSKE, LYBSKE. We measure the execution times of each program's kernel function and compare each redundancy scheme's performance. We run our target programs with two different input sizes provided by the benchmark suite, including *large*, *standard*. We aim to observe threads' parallelism and memory contention for different data sizes.

We evaluate the results by their memory allocation techniques which are large memory and small memory. We discuss the performance differences between the two devices and evaluate the schemes' advantages and disadvantages based on the results.

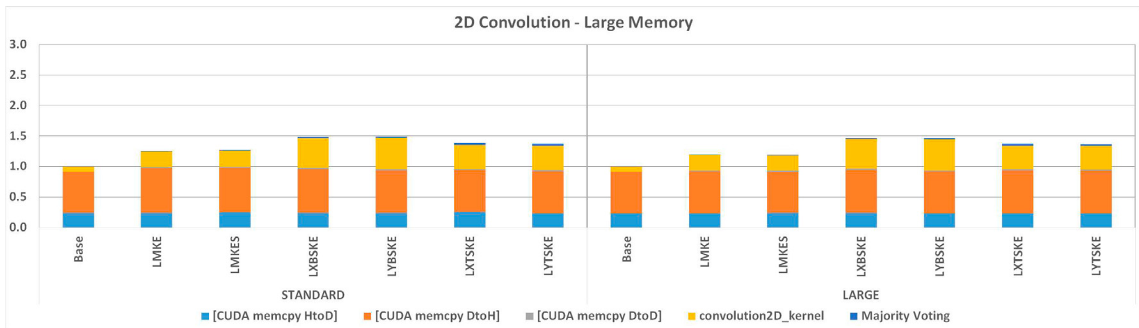
We create two cases to evaluate small memory-based schemes: best-case and worst-case. In the best-case scenario, no error is detected. On the other hand, in the worst-case scenario, an error is detected, and three redundant execution is required.

2D Convolution

2D Convolution's configurations are given in Table 5.3. It has a single kernel function, and the difference in output size is small. Therefore there is no considerable difference in absolute execution time between the configurations.



(a) Normalized Execution Times of 2D Convolution with Large Memory Schemes in Quadro P620.



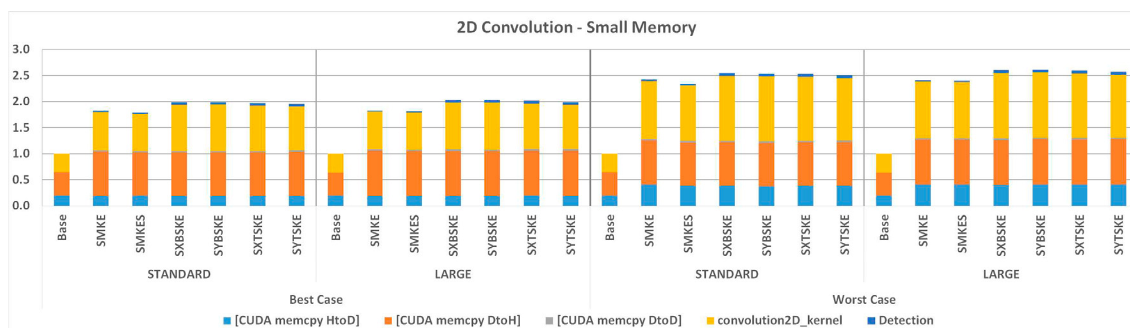
(b) Normalized Execution Times of 2D Convolution with Large Memory Schemes in Tesla K80.

Figure 5.5. Normalized Execution Times of 2D Convolution with Large Memory Schemes in Quadro P620 and Tesla K80.

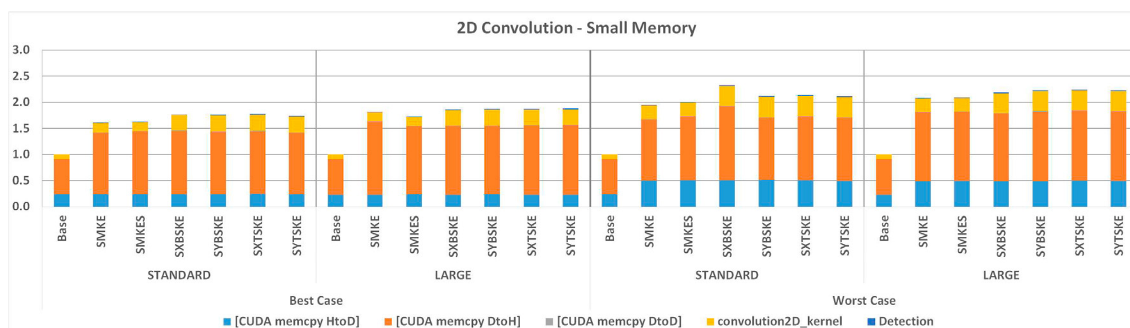
Figure 5.5 demonstrates the normalized execution times of *2D Convolution* with large memory schemes. In the base scheme, memory copy operations take longer than *convolution2D_kernel*. Since the original memory copy operations remain constant and, additional device-to-device memory copy operations are more efficient, the redundant kernel executions make a difference. Besides that, the difference in output size between large and standard configurations is small. Because of this reason, there is no prominent behavior difference between configurations.

Figure 5.5 indicates that MKE-based schemes perform better than SKE-based schemes. The reason behind this behavior is that SKE creates additional threads and blocks. Therefore, handling those causes a significant latency in GPU that causes performance loss. Among SKE-based techniques, thread-based schemes perform better. The reason is that the tripled number of threads is still under the maximum number of threads

per block. Among MKE-based techniques, the LMKES scheme has better performance. The reason is that those kernel executions are nearly overlapped, which is one of our motivations for LMKES.



(a) Normalized Execution Times of 2D Convolution with Small Memory Schemes in Quadro P620.



(b) Normalized Execution times of 2D Convolution with Small Memory Schemes in Tesla K80.

Figure 5.6. Normalized Execution Times of 2D Convolution with Small Memory Schemes in Quadro P620 and Tesla K80.

Figure 5.6 indicates that all schemes show similar behavior to their large memory-based schemes. The main difference is that memory copy operation from device to host increases significantly. That is because we create a checkpoint of the output. In the worst case, along with the device-to-host memory operation, the host-to-device memory operation is increased. That is because we restore output from the checkpoint.

Correlation, Covariance, and Gemm

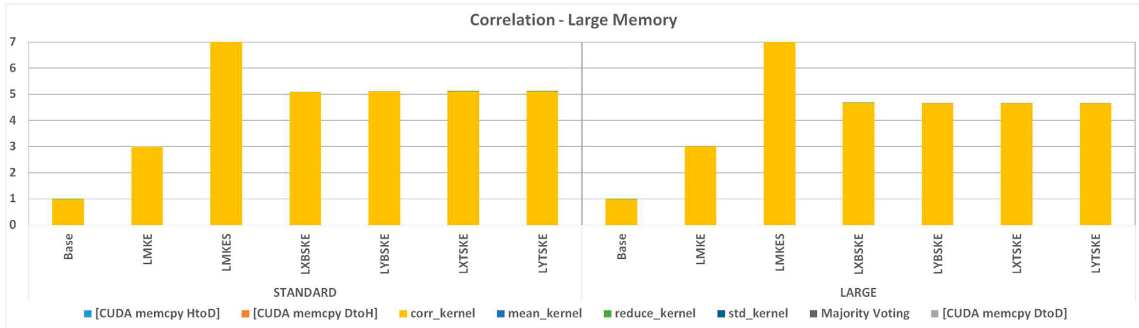
Correlation, Covariance, and *Gemm* have similar characteristics. All the applications are dominated by *corr_kernel, covar_kernel,* and *gemm_kernel,* respectively, as

Table 5.4. Configurations of Correlation, Covariance, and Gemm

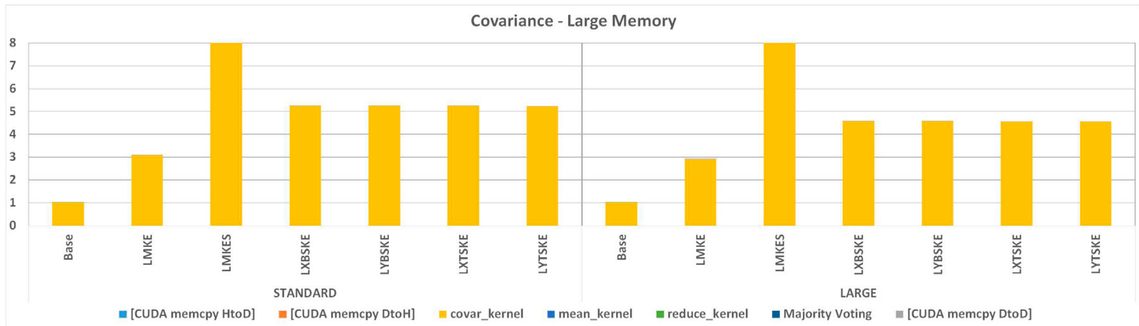
Application	**Kernel**	**Configuration**	**Grid**		**Block**		**Output Size**
			X	*Y*	*X*	*Y*	
Correlation	mean_kernel	Standard	8	1	256	1	2^{11}
		Large	16	1	256	1	2^{12}
	std_kernel	Standard	8	1	256	1	2^{11}
		Large	16	1	256	1	2^{12}
	reduce_kernel	Standard	64	256	32	8	2^{22}
		Large	128	512	32	8	2^{24}
	corr_kernel	Standard	8	1	256	1	2^{22}
		Large	16	1	256	1	2^{24}
Covariance	mean_kernel	Standard	8	1	256	1	2^{11}
		Large	16	1	256	1	2^{12}
	reduce_kernel	Standard	64	256	32	8	2^{22}
		Large	128	512	32	8	2^{24}
	covar_kernel	Standard	8	1	256	1	2^{22}
		Large	16	1	256	1	2^{24}
Gemm	Gemm_kernel	Standard	16	64	32	8	2^{18}
		Large	32	128	32	8	2^{20}

seen in Figure 5.7. Table 5.4 shows that their grid size increases as the output's size increases while block size does not change. We can see that while the X dimension of the grid of *Correlation's* and *Covariance's* dominant kernels are more intense, the Y dimension of the grid of *Gemm*'s kernel. We should remember that Correlation and Covariance applications use more memory than *Gemm*.

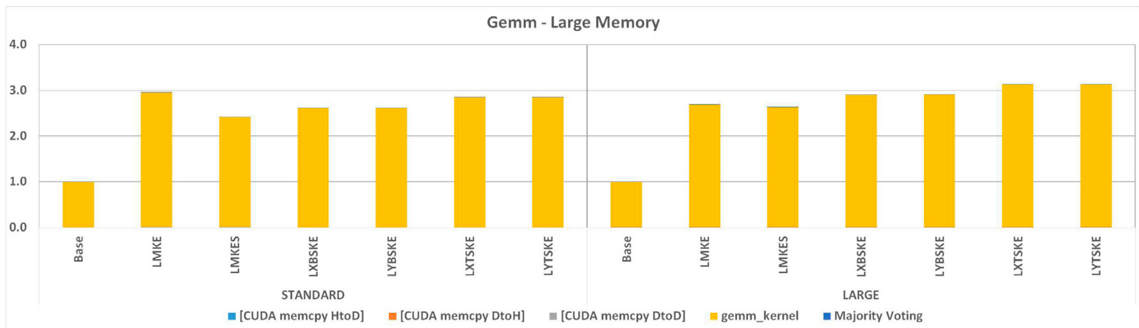
The common characteristic of the applications is that three of them are dominated by one specific kernel. This situation leads to similar results. While *Correlation* and *Covariance* are memory-intensive applications, *Gemm* is a compute-intensive ap-



(a) Normalized Execution Times of Correlation with Large Memory Schemes in Quadro P620.



(b) Normalized Execution Times of Covariance with Large Memory Schemes in Quadro P620.

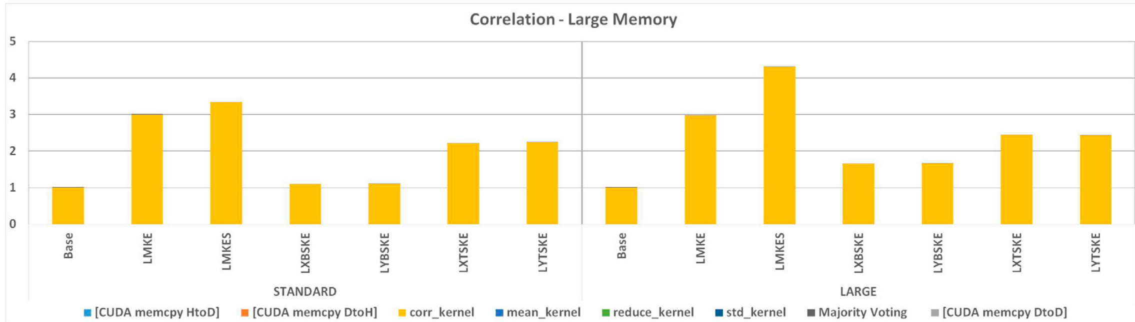


(c) Normalized Execution Times of Gemm with Large Memory Schemes in Quadro P620.

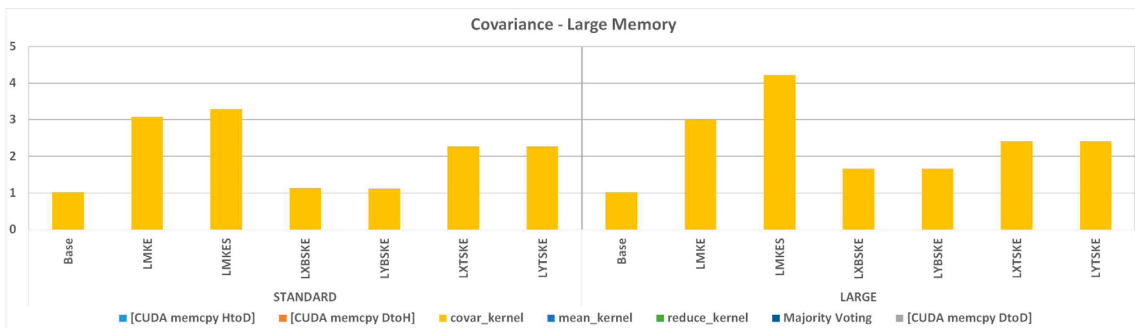
Figure 5.7. Normalized Execution Times of Correlation, Covariance, and Gemm with Large Memory Schemes in Quadro P620.

plication. Therefore, we expect that *Gemm* perform better in SKE-based schemes and *Correlation* and *Covariance* perform better in MKE-based schemes. However, as shown in Figures 5.7a and 5.7b, both Correlation and Covariance applications perform poorly in the LMKES scheme with Quadro P620. The reasons are that handling the streams creates a fair latency, and streams put pressure on the memory. Hence, it affects applications'

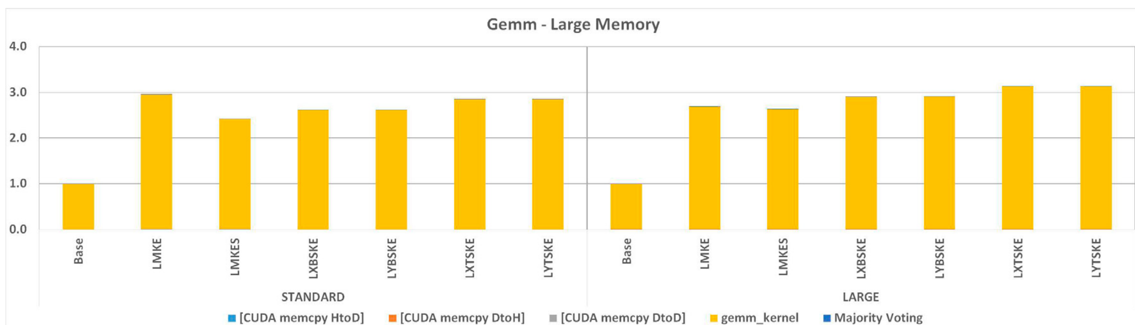
performances. We also observe similar behavior in Tesla K80 as seen in Figures 5.8a and 5.8b. LMKES perform worse than LMKE. However, SKE-based schemes perform well. The reasons are that Tesla K80 has more CUDA cores and global memory than Quadro P620 as indicated in Table 5.1. Therefore, additional streams, blocks, and threads do not put as much pressure on Tesla K80 as on Quadro P620.



(a) Normalized Execution Times of Correlation with Large Memory Schemes in Tesla K80.



(b) Normalized Execution Times of Covariance with Large Memory Schemes in Tesla K80.

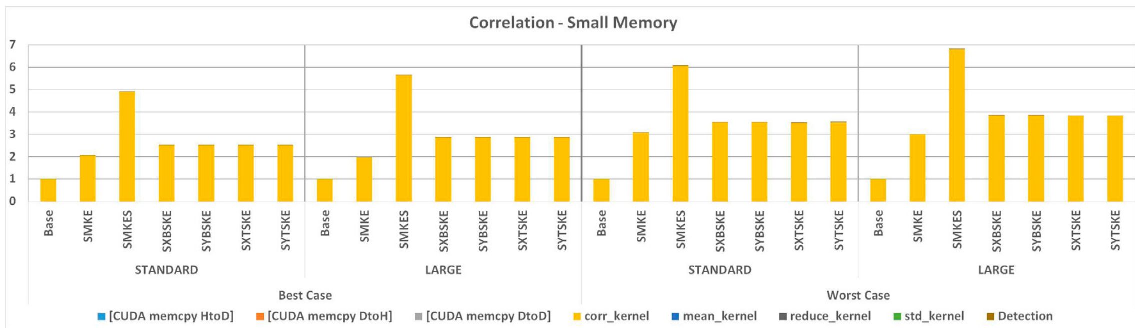


(c) Normalized Execution Times of Gemm with Large Memory Schemes in Tesla K80.

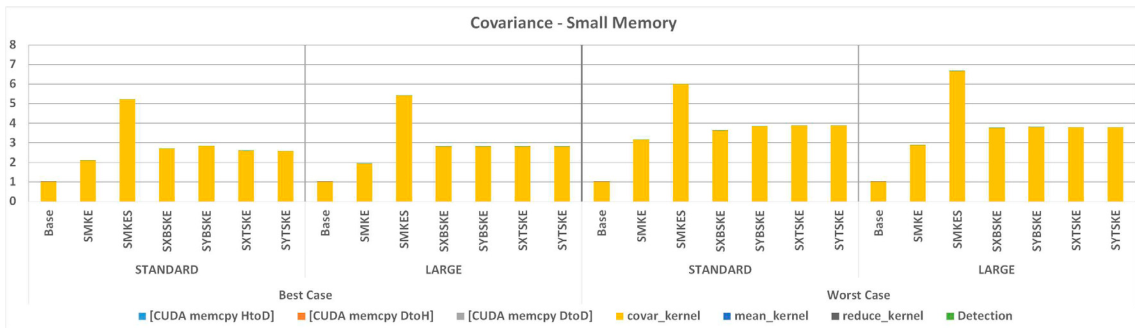
Figure 5.8. Normalized Execution Times of Correlation, Covariance, and Gemm with Large Memory Schemes in Tesla K80.

As seen in the Figures 5.7c and 5.8c, *Gemm* shows a decent performance increase in LMKES compared to LMKE. The reason is that since it is not memory intense, additional streams do not put pressure on the memory. Besides that, we also observe that the performance difference between LMKE and LMKES decreases as the output size increases. It also proves our conclusion. Among the SKE-based, L*TSKE schemes perform better than L*BSKE schemes. The reason is that block organization of the *gemm_kernel* is more intense than thread organization.

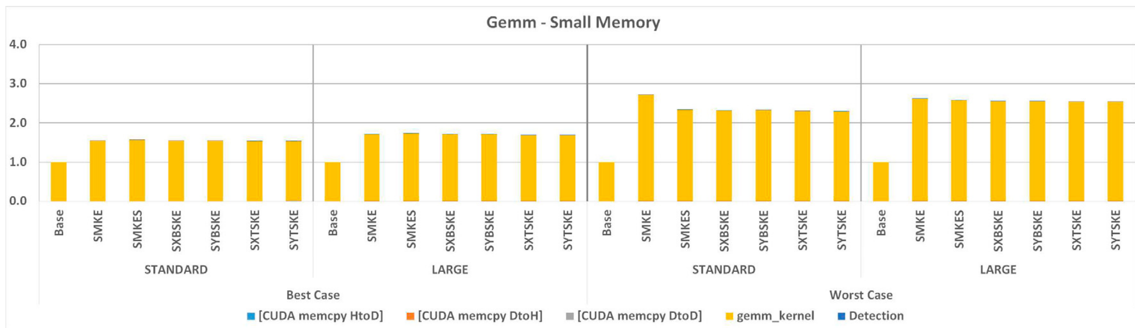
Those applications show similar results in small memory schemes. Even though, additional memory copy operations create latency, it is negligible as seen in Figures 5.9 and 5.10. However, in the best case, all applications slightly increase block-based SKE techniques. Hence, doubling the number of blocks does not cause significant overhead. We can say that if it is likely that execution will result error-free, using those schemes is beneficial. According to Figures 5.9c and 5.10c, additional memory operations are also negligible because output size is small in Gemm. Even in the worst case, it performs better than its corresponding large memory-based schemes. If there is no error, the total execution time reduces by around 30%. Therefore, it is beneficial to use small memory in either way.



(a) Normalized Execution Times of Correlation with Small Memory Schemes in Quadro P620.

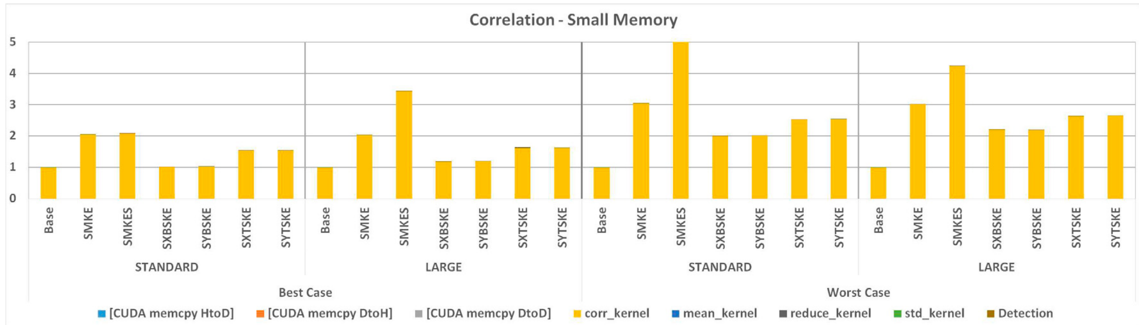


(b) Normalized Execution Times of Covariance with Small Memory Schemes in Quadro P620.

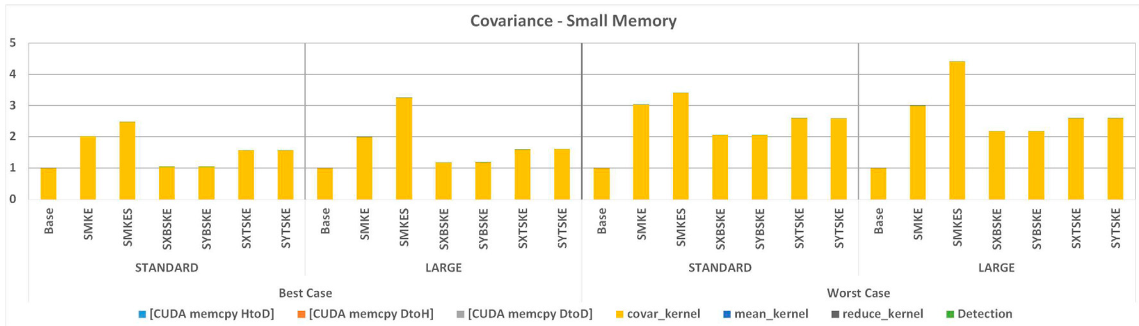


(c) Normalized Execution Times of Gemm with Small Memory Schemes in Quadro P620.

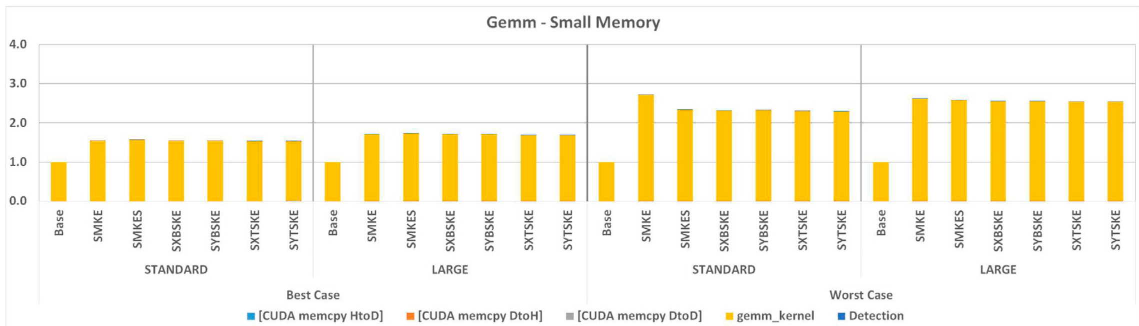
Figure 5.9. Normalized Execution Times of Correlation, Covariance, and Gemm with Small Memory Schemes in Quadro P620.



(a) Normalized Execution Times of Correlation with Small Memory Schemes in Tesla K80.



(b) Normalized Execution Times of Covariance with Small Memory Schemes in Tesla K80.



(c) Normalized Execution Times of Gemm with Small Memory Schemes in Tesla K80.

Figure 5.10. Normalized Execution Times of Correlation, Covariance, and Gemm with Small Memory Schemes in Tesla K80.

3D Convolution, Fdtd2d, and Gramschmidt

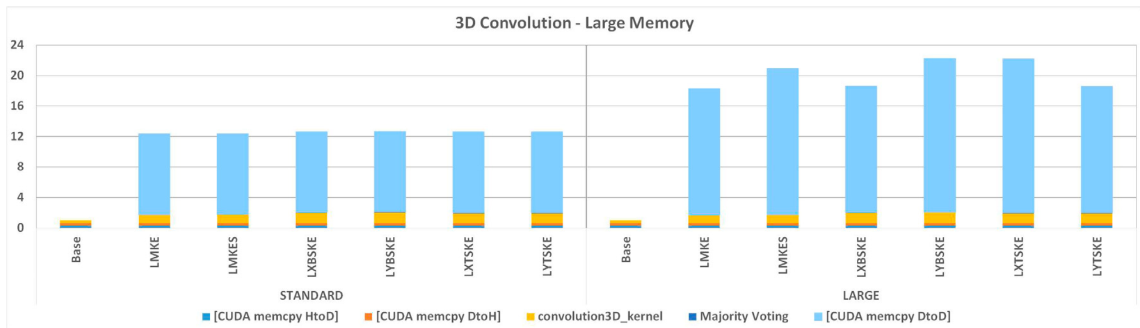
The kernels of *3D Convolution, Fdtd2d,* and *Gramschmidt*, are executed in a loop. This attribute leads to many function calls, and every redundant function call comes with additional memory copy operations. As seen in Figures 5.11 and 5.12, additional memory operations causes significant latency. Besides that, *3D Convolution* is the most affected

Table 5.5. Configurations of 3D Convolution, Fdtd2d and Gramschmidt

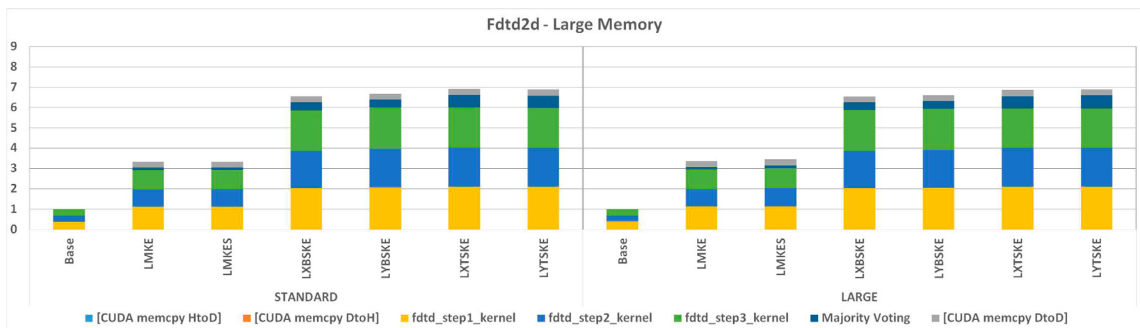
Application	**Kernel**	**Configuration**	**Grid**		**Block**		**Output Size**
			X	*Y*	*X*	*Y*	
3D Convolution	convolution3D_kernel	Standard	8	32	32	8	2^{24}
		Large	12	48	32	8	2^{26}
Fdtd2d	fdtd_step1_kernel	Standard	64	256	32	8	2^{22}
		Large	128	512	32	8	2^{24}
	fdtd_step2_kernel	Standard	64	256	32	8	2^{22}
		Large	128	512	32	8	2^{24}
	fdtd_step3_kernel	Standard	64	256	32	8	2^{22}
		Large	128	512	32	8	2^{24}
Gramschmidt	gramschmidt_kernel1	Standard	1	1	256	1	2^{22}
		Large	1	1	256	1	2^{24}
	gramschmidt_kernel2	Standard	8	1	256	1	2^{22}
		Large	16	1	256	1	2^{24}
	gramschmidt_kernel3	Standard	8	1	256	1	2^{22}
		Large	16	1	256	1	2^{24}

of all the apps. This is because, as seen in Table 5.5, 3D Convolution deals with slightly larger output, and *convolution3D_kernel* does not take as much time as other kernels. Therefore, additional memory copy operations cause a more significant increase in 3D Convolution compared to Fdtd2d and Gramschmidt.

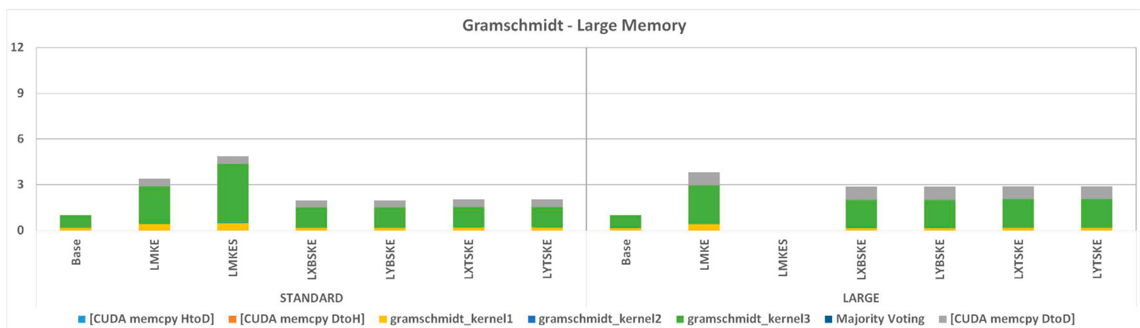
Unlike 3D Convolution and Gramschmidt, the number of iterations does not change as configuration changes in Fdtd2d. Therefore, the number of function calls does not increase with configuration. However, the number of blocks increases as seen in Table 5.5. Hence, handling the tripled number of blocks becomes a significant concern. Therefore, Fdtd2d performs better with MKE schemes compared to SKE schemes as shown in Figures 5.11b and 5.12b.



(a) Normalized Execution Times of 3D Convolution with Large Memory Schemes in Quadro P620.



(b) Normalized Execution Times of Fdtd2d with Large Memory Schemes in Quadro P620.

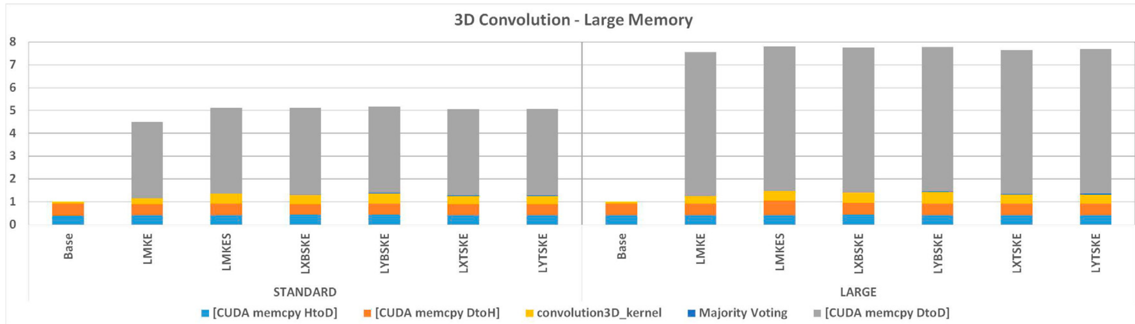


(c) Normalized Execution Times of Gramschmidt with Large Memory Schemes in Quadro P620.

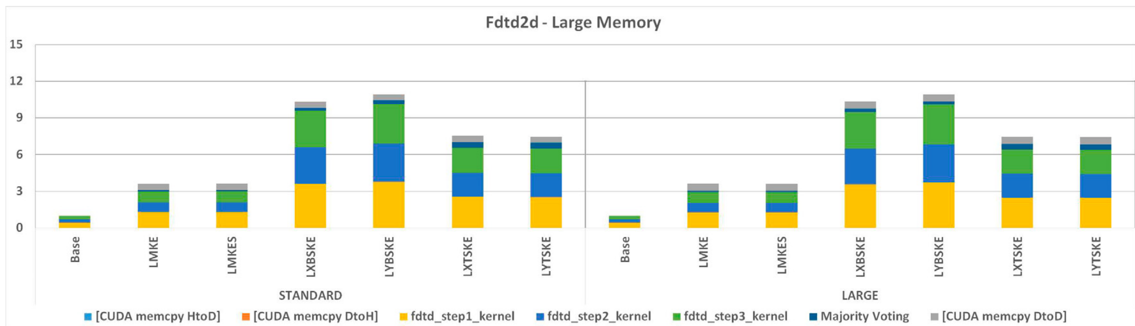
Figure 5.11. Normalized Execution Times of 3D Convolution, Fdtd2d, and Gramschmidt with Large Memory Schemes in Quadro P620.

Along with the number of iterations, the number of blocks also increases as configuration changes in Gramschmidt. However, as shown in Table 5.5, the kernel's number of blocks is tiny. Therefore, it is easier to handle, and that leads to SKE schemes performing better than MKE schemes, as shown in Figures 5.11c and 5.12c. Another significant

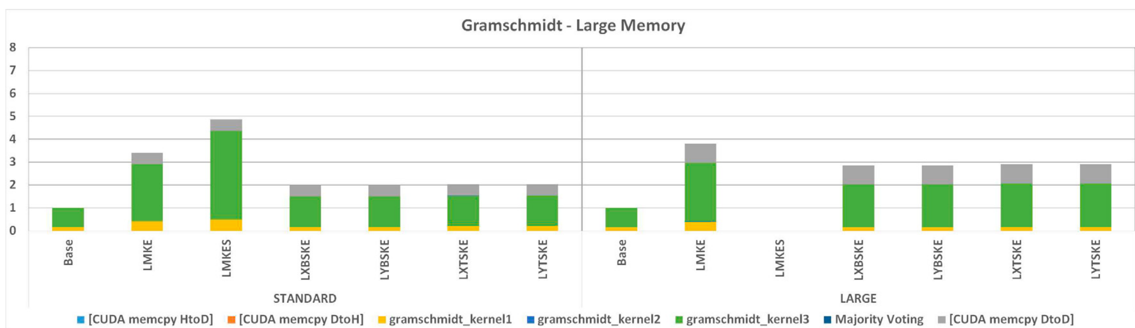
result is that LMKES does not work with either GPU while LMKE works. Compare to *gramschmidt_kernel1* and *gramschmidt_kernel2*, *gramschmidt_kernel3* takes more time and we create streams for all the kernels. Therefore, the overhead caused by streams overwhelms its advantages, leading the application not to execute properly.



(a) Normalized Execution Times of 3D Convolution with Large Memory Schemes in Tesla K80.

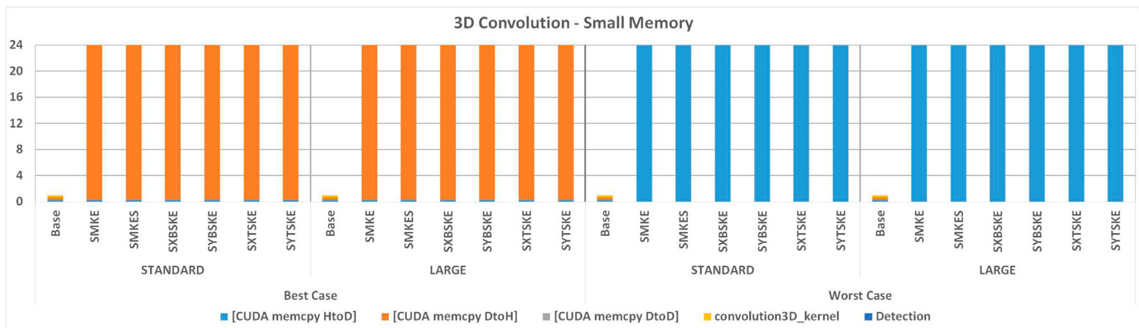


(b) Normalized Execution Times of Fdtd2d with Large Memory Schemes in Tesla K80.

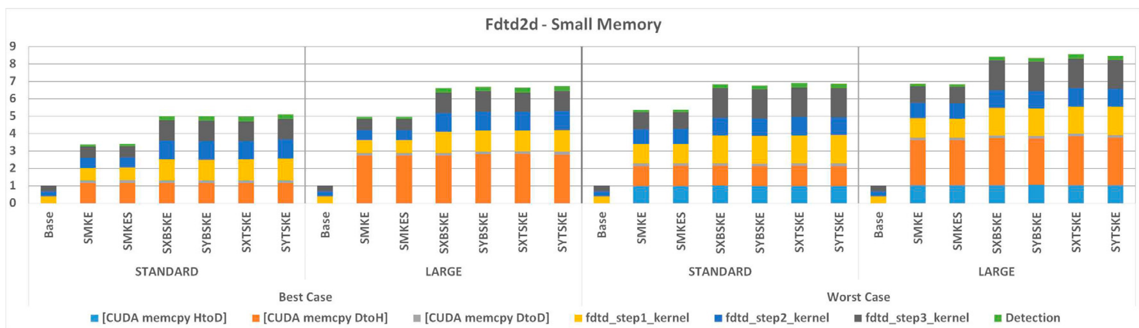


(c) Normalized Execution Times of Gramschmidt with Large Memory Schemes in Tesla K80.

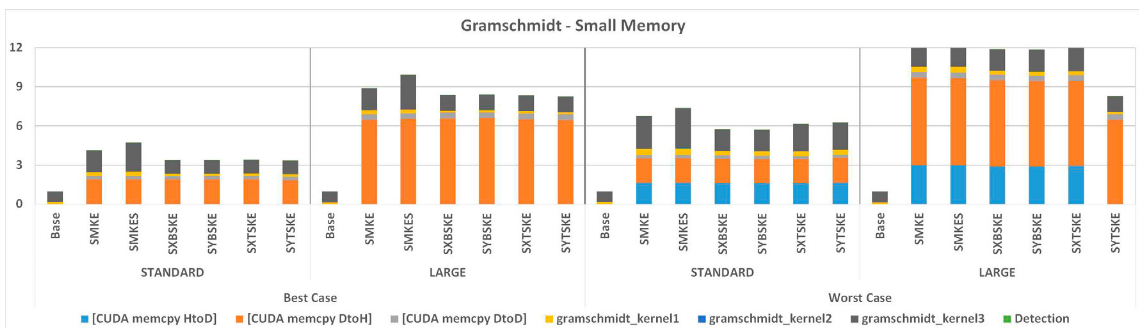
Figure 5.12. Normalized Execution Times of 3D Convolution, Fdtd2d, and Gramschmidt with Large Memory Schemes in Tesla K80.



(a) Normalized Execution Times of 3D Convolution with Small Memory Schemes in Quadro P620.



(b) Normalized Execution Times of Fdtd2d with Small Memory Schemes in Quadro P620.

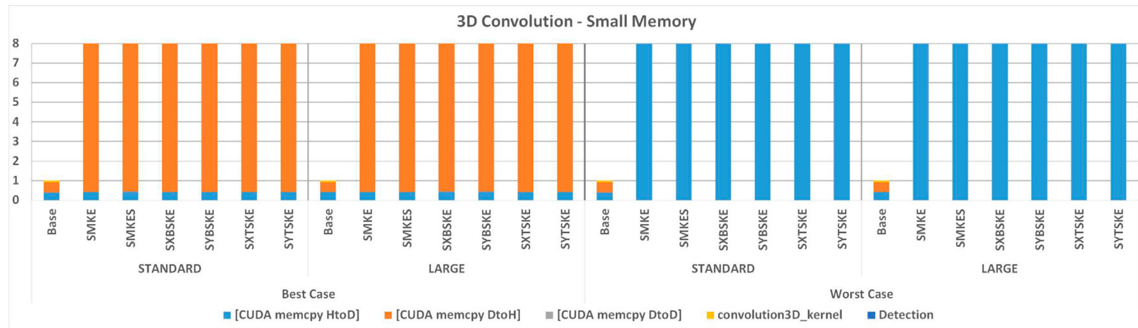


(c) Normalized Execution Times of Gramschmidt with Small Memory Schemes in Quadro P620.

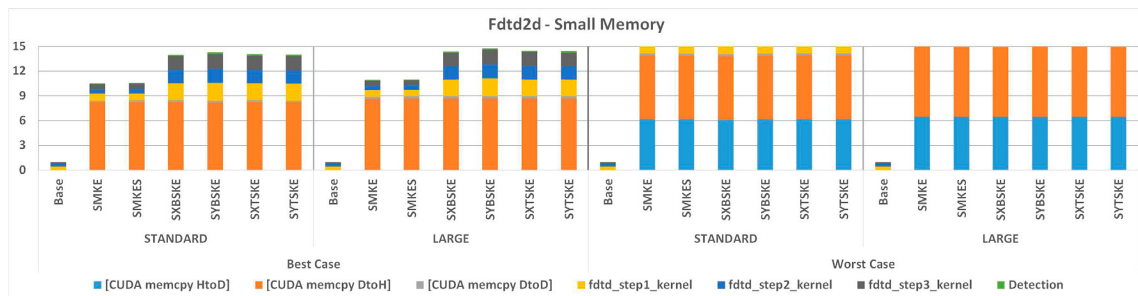
Figure 5.13. Normalized Execution Times of 3D Convolution, Fdtd2d, and Gramschmidt with Small Memory Schemes in Quadro P620.

As mentioned earlier, 3D Convolution, Fdtd2d, and Gramschmidt show similar results in large memory schemes. This situation is the same for small memory schemes. All three application suffer from great performance loss as shown in Figures 5.13 and 5.14. All kernels of three applications are executed in a loop, leading to memory oper-

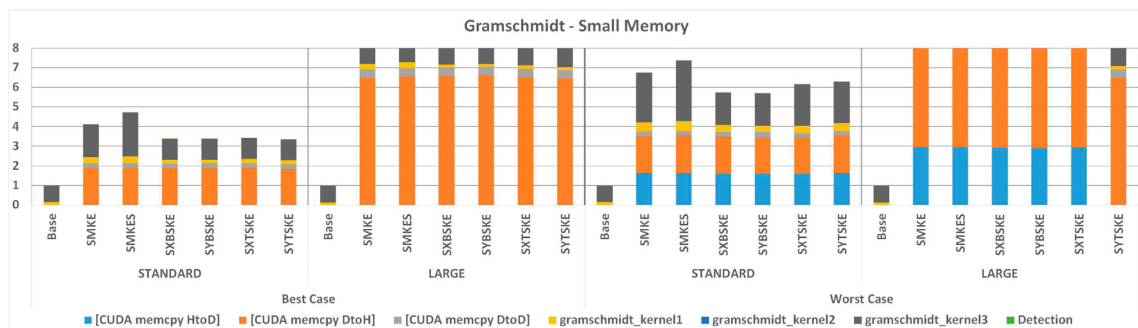
ation from device-to-host in every iteration. Especially in the worst case, host-to-device memory copy operations are also added. Therefore, it results in great performance loss. Additionally, since the number of iterations does not change based on the configurations in Fdtd2d, the number of memory operations also does not change. However, the amount of data is increased; therefore, the latency of memory operations becomes noticeable.



(a) Normalized Execution Times of 3D Convolution with Small Memory Schemes in Tesla K80.



(b) Normalized Execution Times of Fdtd2d with Small Memory Schemes in Tesla K80.



(c) Normalized Execution Times of Gramschmidt with Small Memory Schemes in Tesla K80.

Figure 5.14. Normalized Execution Times of 3D Convolution, Fdtd2d, and Gramschmidt with Small Memory Schemes in Tesla K80.

Our motivation behind small memory schemes is to ensure that applications requiring significant memory are protected. An additional remark for Gramschmidt is SMKES works unlike LMKES as shown in Figures 5.13c and 5.14c.

3mm and Bicg

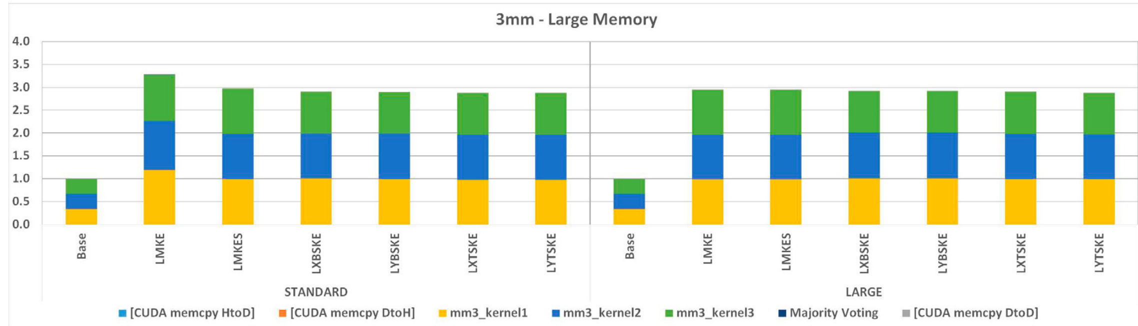
Table 5.6. Configurations of 3mm and Bicg

Application	Kernel	Configuration	Grid		Block		Output Size
			X	*Y*	*X*	*Y*	
3mm	mm3_kernel1	Standard	16	64	32	8	2^{18}
		Large	32	128	32	8	2^{20}
	mm3_kernel2	Standard	16	64	32	8	2^{18}
		Large	32	128	32	8	2^{20}
	mm3_kernel3	Standard	16	64	32	8	2^{18}
		Large	32	128	32	8	2^{20}
Bicg	bicg_kernel1	Standard	16	1	256	1	2^{12}
		Large	32	1	256	1	2^{13}
	bicg_kernel2	Standard	16	1	256	1	2^{12}
		Large	32	1	256	1	2^{13}

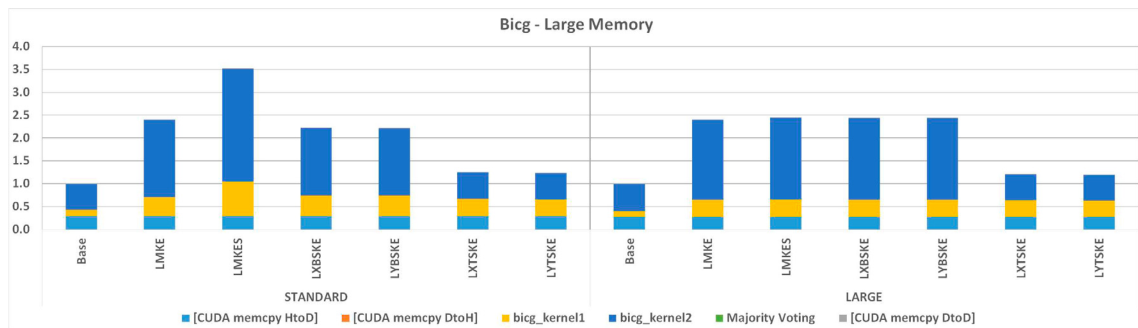
3mm and *Bicg* have similar characteristics. Both applications execute a small number of functions with relatively small outputs. As shown in Figure 5.15a, SKE schemes perform slightly better than MKE schemes with Large Memory Schemes in Quadro P620. As indicated in Table 5.6, the tripled number of blocks and threads does not exceed the maximum amount. Since MKE schemes increase the number of function calls and streams require additional handling, SKE schemes are expected to perform better.

According to Figure 5.15b, the difference between LMKE and LMKES seems significant. However, the kernels' executions take very little time, and the original host-to-device memory copy operations take most of the execution time in the base scheme; hence creating the streams and handling them takes more time than the kernel's execution. We can support this observation by observing the difference between LMKE and

LMKES as output size increases. In the large configuration, the difference between LME and LMKES is much less than the difference in the small configuration. Additionally, as shown in Figure 5.16b, the difference between LMKE and LMKES is reduced in Tesla K80 compared to Quadro P620. This is because Tesla K80 has a lower clock rate. Therefore, it can exploit the stream feature better than Quadro P620 in this case.



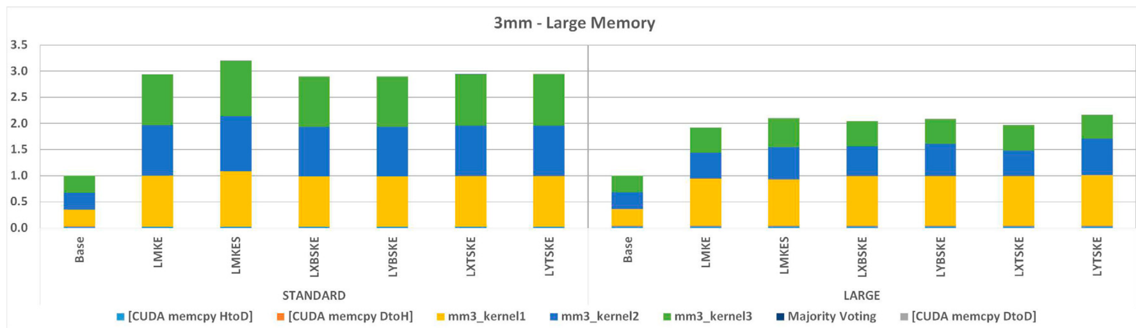
(a) Normalized Execution Times of 3mm with Large Memory Schemes in Quadro P620.



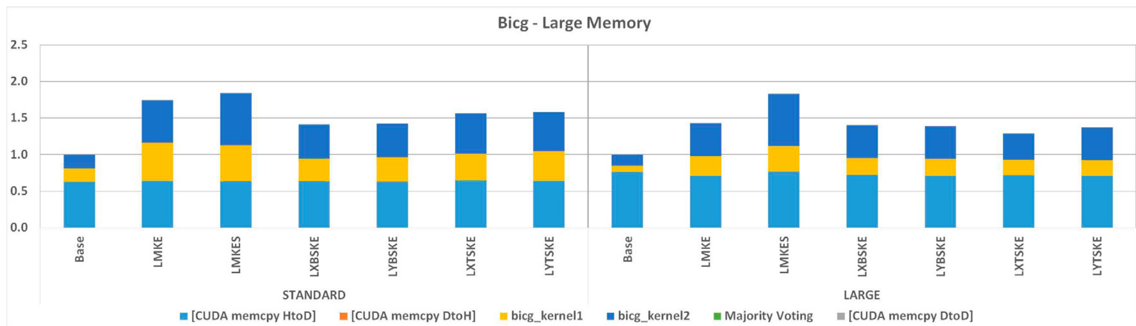
(b) Normalized Execution Times of Bicg with Large Memory Schemes in Quadro P620.

Figure 5.15. Normalized Execution Times of 3mm and Bicg with Large Memory Schemes in Quadro P620.

Since the kernels in *3mm* do not deal with big arrays, changes in memory operations are not noticeable. Therefore, when in the worst case, it performs similarly to its corresponding large memory-based scheme as seen in Figures 5.17a and 5.18a. In such cases, it is beneficial to use low memory-based techniques.



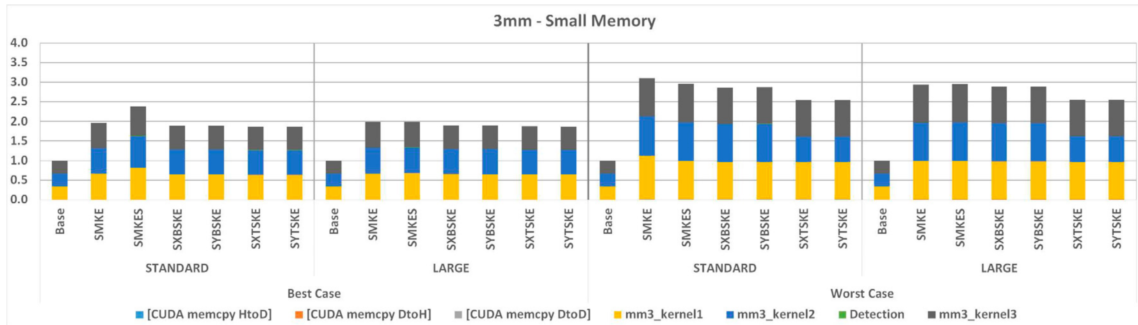
(a) Normalized Execution Times of 3mm with Large Memory Schemes in Tesla K80.



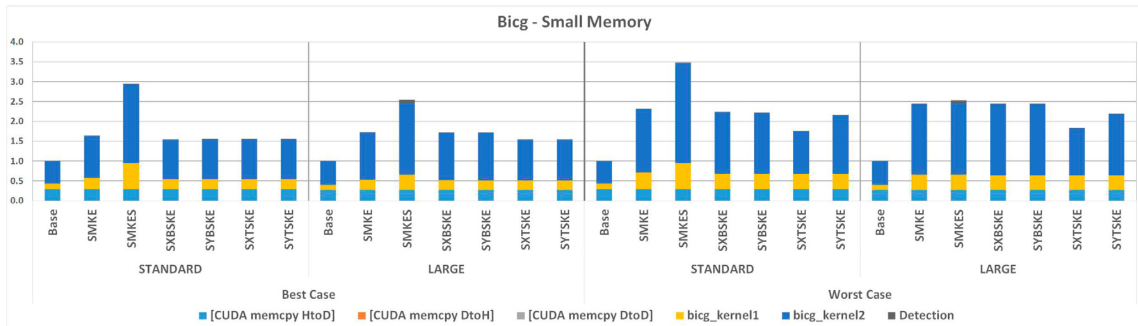
(b) Normalized Execution Times of Bicg with Large Memory Schemes in Tesla K80.

Figure 5.16. Normalized Execution Times of 3mm, and Bicg with Large Memory Schemes in Tesla K80.

According to Figures 5.17b and 5.18b, bicg shows similar characteristic to 3mm. However, since kernels of the bicg's executions take little time and the outputs' size is small, the changes are not noticeable. As excepted, in best cases, results converge to two as in large memory, result converges to three. The worst case can deduce a similar observation.

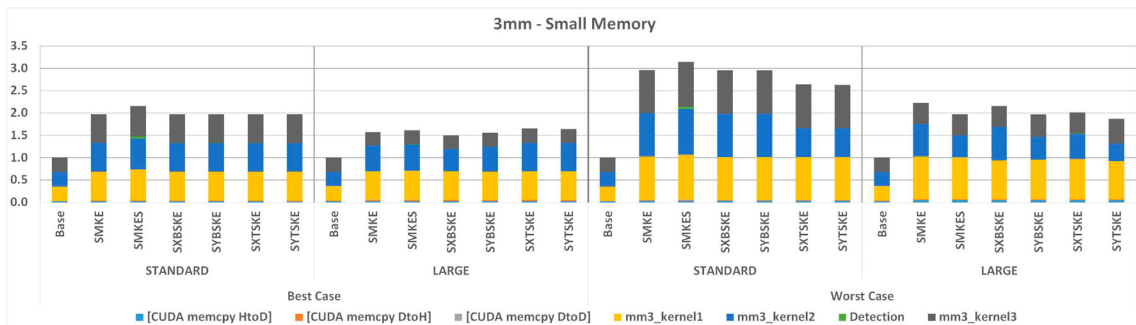


(a) Normalized Execution Times of 3mm with Small Memory Schemes in Quadro P620.

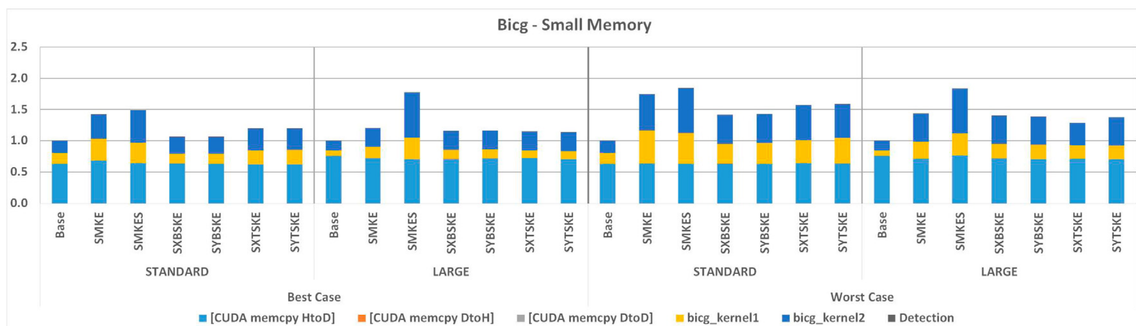


(b) Normalized Execution Times of Bicg with Small Memory Schemes in Quadro P620.

Figure 5.17. Normalized Execution Times of 3mm and Bicg with Small Memory Schemes in Quadro P620.



(a) Normalized Execution Times of 3mm with Small Memory Schemes in Tesla K80.



(b) Normalized Execution Times of Bicg with Small Memory Schemes in Tesla K80.

Figure 5.18. Normalized Execution Times of 3mm, and Bicg with Small Memory Schemes in Tesla K80.

CHAPTER 6

CONCLUSION AND FUTURE WORKS

In this thesis, we present our compiler-managed fault tolerance techniques for GPGPUs. We offer twelve different schemes to exploit existing GPU and overcome memory limitations.

Our primary scheme, *MKE*, offers protection by executing the annotated function redundantly. Its improved version, *MKES*, also executes the annotated function redundantly. However, it uses CUDA streams to exploit the massively parallel architecture of NVIDIA GPUs.

Our latter schemes are grouped into *SKE*. It executes the annotated function by increasing the number of threads or blocks. In this way, it offers redundant execution with a single function call to exploit the numerous cores of GPUs.

Our schemes are also grouped by their memory allocation techniques: large memory and small memory. Large memory schemes use three redundant outputs in a given time. However, the three redundant outputs might fit in the global memory of GPUs with memory limitations. Therefore, we create small memory schemes. Our small memory schemes use the host's memory along with the global memory of the GPU. Those schemes trade-off performance for lower global memory usage.

Based on our experiments, our conclusions are as follows:

1. Partial redundancy approach trade-offs acceptable error for better performance. It is suitable for domains such as machine learning and computer vision.

2. Different applications show different performance patterns based on our techniques.

3. Our MKE-based techniques perform better if the number of blocks or threads exceeds the maximum amounts, and our SKE-based techniques perform better if the GPU has the resources for executing them in parallel.

4. Our large memory-based techniques perform better if the application deal with large data.

5. Our small memory-based techniques perform better if the application deal with small data. In exchange for performance loss, it can also perform fault coverage in huge data.
6. We create small memory-based techniques to provide protection even though the GPU has memory limitations. We achieved this.

6.1. Future Works

Our work shows decent results. However, there is still room for improvement. The potential improvements are given below for future works.

1. **Merge with the Fault Injection Tool**
   Our custom compiler and the fault injection tool can be merged to create a framework. It can give programmer suggestions on which scheme to use and which function to replicate.
2. **Extending the Sphere of Replication (SOR)**
   SOR is a critical concept for RMT. SOR defines the components included in the redundant execution. For our schemes, SOR is limited to the output. We can extend the SOR to cover multiple outputs and inputs.
3. **Output Limitations**
   Our techniques replicate a single output if it is listed last in the function call. This limitation does not fit every application. We wish to remove this limitation.
4. **GPU and Programming Model Limitations**
   Our techniques support only NVIDIA GPUs and CUDA Programming. It can be extended to cover other programming models and GPUs, such as OpenCL and AMD GPUs.

REFERENCES

Aamodt, T. M., W. W. L. Fung, and T. G. Rogers (2018). General-purpose graphics processor architectures. *Synthesis Lectures on Computer Architecture 13*(2), 1–140.

Alcaide, S., L. Kosmidis, C. Hernandez, and J. Abella (2021). Achieving diverse redundancy for gpu kernels. *IEEE Transactions on Emerging Topics in Computing 10*(2), 618–634.

Bohman, M., B. James, M. J. Wirthlin, H. Quinn, and J. Goeders (2019). Microcontroller compiler-assisted software fault tolerance. *IEEE Transactions on Nuclear Science 66*(1), 223–232.

Chang, C.-K., G. Li, and M. Erez (2019). Evaluating compiler ir-level selective instruction duplication with realistic hardware errors. In *2019 IEEE/ACM 9th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*, pp. 41–49.

Chen, J., H. Li, S. Li, X. Liang, P. Wu, D. Tao, K. Ouyang, Y. Liu, K. Zhao, Q. Guan, and Z. Chen (2018). Fault tolerant one-sided matrix decompositions on heterogeneous systems with gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18.

Colaboratory-Frequently Asked Questions (2022). Colaboratory-frequently asked questions. Last accessed 3 January 2022.

Didehban, M. and A. Shrivastava (2016). Nzdc: A compiler technique for near zero silent data corruption. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, New York, NY, USA. Association for Computing Machinery.

Feng, S., S. Gupta, A. Ansari, and S. Mahlke (2010). Shoestring: probabilistic soft error reliability on the cheap. *ACM SIGARCH Computer Architecture News 38*(1), 385–396.

Grauer-Gray, S., L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos (2012, 5). Auto-tuning a high-level language targeted to GPU codes. IEEE.

Gupta, M., D. Lowell, J. Kalamatianos, S. Raasch, V. Sridharan, D. Tullsen, and R. Gupta (2017). Compiler techniques to reduce the synchronization overhead of gpu redundant multithreading. In *Design Automation Conference (DAC)*.

HajiRassouliha, A., A. J. Taberner, M. P. Nash, and P. M. Nielsen (2018). Suitability of recent hardware accelerators (dsps, fpgas, and gpus) for computer vision and image processing algorithms. *Signal Processing: Image Communication 68*, 101–119.

Hamming, R. W. (1950). Error detecting and error correcting codes. *Bell System Technical Journal 29*, 147–160.

Harris, M. (2022, Jan). How to overlap data transfers in cuda c/c++.

Kalra, C., F. Previlon, N. Rubin, and D. Kaeli (2020). Armorall: Compiler-based resilience targeting gpu applications. *ACM Transactions on Architecture and Code Optimization (TACO) 17*(2), 1–24.

Kaya, E., Ömer Faruk Karadaş, and I. Öz (2021). Evaluating performance and reliability of selective redundant multithreading for gpgpu applications. In *1st Workshop on Connecting Education and Research Communities for an Innovative Resource Aware Society (CERCIRAS), Novi Sad, Serbia*.

Kim, S. and A. Somani (1999). Area efficient architectures for information integrity in cache memories. In *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, pp. 246–255.

Kim, Y., R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu (2014). Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pp. 361–372. IEEE Press.

Kirk, D. B. and W. H. Wen-Mei (2016). *Programming massively parallel processors: a hands-on approach* (Third ed.). Morgan kaufmann.

Koren, I. and C. M. Krishna (2020). *Fault-tolerant systems*. Morgan Kaufmann.

Lattner, C. (2012). *LLVM*, Volume 1, pp. 155–171. lulu.com.

Lattner, C. and V. Adve (2004). Llvm: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86.

Lee, V. W., C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey (2010). Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, New York, NY, USA, pp. 451–460. Association for Computing Machinery.

Leng, J., T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi (2013). Gpuwattch: Enabling energy optimizations in gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, New York, NY, USA, pp. 487–498. Association for Computing Machinery.

Lingqi Zhang, M. W. and S. Matsuoka (2019). Understanding the overheads of launching cuda kernels. In *International Conference on Parallel Processing (ICPP)*.

LLVM Project (2022). User guide for nvptx back-end.

Mahmoud, A., S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler (2018). Optimizing software-directed instruction replication for gpu error detection. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18.

Martin, E. (2012). *LLVM*, Volume 1, pp. 33–43. lulu.com.

Memik, G., M. T. Kandemir, and O. Ozturk (2005). Increasing register file immunity to transient errors. In *Design, Automation and Test in Europe*, pp. 586–591. IEEE.

Mittal, S. and J. S. Vetter (2014). A survey of methods for analyzing and improving gpu energy efficiency. *ACM Computing Surveys (CSUR) 47*(2), 1–23.

Mukherjee, S. (2011). *Architecture design for soft errors*. Morgan Kaufmann.

Oz, I. and S. Arslan (2019). A survey on multithreading alternatives for soft error fault tolerance. *ACM Computing Surveys (CSUR) 52*(2), 1–38.

Öz, I. and Ö. F. Karadaş (2022). Regional soft error vulnerability and error propagation analysis for gpgpu applications. *The Journal of Supercomputing 78*(3), 4095–4130.

Portet, S. A., L. Kosmidis, C. Hernandez, and J. Abella (2020). Software-only triple diverse redundancy on gpus for autonomous driving platforms. In *2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, pp. 82–88. IEEE.

Profiler User's Guide (2022, May). Profiler user's guide.

Reis, G., J. Chang, N. Vachharajani, R. Rangan, and D. August (2005). Swift: software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*, pp. 243–254.

Schlegel, D. (2015). Deep machine learning on gpu. *University of Heidelber-Ziti 12*.

Sugihara, M., T. Ishihara, and K. Murakami (2007). Task scheduling for reliable cache architectures of multiprocessor systems. In *2007 Design, Automation & Test in Europe Conference & Exhibition*, pp. 1–6. IEEE.

Tabkhi, H. and G. Schirner (2012). Application-specific power-efficient approach for reducing register file vulnerability. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 574–577. IEEE.

Tavarageri, S., S. Krishnamoorthy, and P. Sadayappan (2014). Compiler-assisted detection of transient memory errors. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 204–215.

Wadden, J., A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron (2014). Real-world design and evaluation of compiler-managed gpu redundant multithreading. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 73–84.

Xu, J., Q. Tan, and H. Zhou (2011). Scheduling instructions for soft errors in register files. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pp. 305–312. IEEE.

Yang, L., B. Nie, A. Jog, and E. Smirni (2021). Enabling software resilience in gpgpu applications via partial thread protection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1248–1259.

Yitbarek, S. F. and T. Austin (2018). Reducing the overhead of authenticated memory encryption using delta encoding and ecc memory. In *Proceedings of the 55th Annual Design Automation Conference*, pp. 1–6.

Zhang, W., S. Gurumurthi, M. T. Kandemir, and A. Sivasubramaniam (2003). Icr: In-cache replication for enhancing data cache reliability. In *DSN*, pp. 291–300.