POLITECNICO DI TORINO

Master's Degree in Embedded Systems

Master's Thesis

Reliability issues in GPGPUs

Supervisors:
Prof. Sonza Reorda Matteo
Prof. Sterpone Luca
Dr. Rodriguez Condia Josie Esteban
Dr. Du Boyang

Candidate:
Penaglia Chiara
ID: S246289

December 2019

Abstract

The present work discusses different approaches for implementing software-based redundancy schemes using the FlexGrip open-source GPGPU model to improve GPGPU reliability.

Most GPGPUs do not feature hardware support for error detection, and in a device such as a GPGPU a corrupted result could be unacceptable, as applications such as machine vision rely on the correctness of the processed image. A fault could occur at any time during the operation of the device, and it is critical that it is either masked or detected. Therefore, improving the reliability of GPGPUs using software redundancy seems to be the only way to avoid errors.

In this thesis work several approaches for matrix multiplication were produced, recording and analysis the performance of each. The three approaches differ in the method by which they guarantee the correct result: The first case is Duplication With Comparison (DWC) which implies repeatedly performing operations and comparing the results, storing the result in memory only if the two match. The second method is Triple Modular Redundancy (TMR). It is based on the triplication of resources and a voter who establishes by a majority which element is the correct one. The last method studied is Algorithm Based Fault Tolerance (ABFT) which through comparisons identifies in which cell the error occurred and corrects it.

Each code was tested on the FlexGrip model after the injection of static faults inside the register file of each streaming multiprocessor. The expected result of each program obtained in simulation - the "*golden output*" - was compared to the same result in presence of injected static faults.

Results were finally collected and the fault coverage analysed, along with the time required and memory space. Future tests may be performed with different fault models, such as transient or delay faults, since the behaviour of the circuit would vary unpredictably.[4] [8]

Contents

List of Figures

Chapter 1

Introduction

The present introduction will provide the reader with a summary of the contents of this work of thesis.

Summary

Nowadays, GPUs have a great impact on our life, from the most trivial areas, such as video games, to the most critical such as aeronautics or medicine. Generally, GPU usage is desirable in all fields that require processing very large amounts of pictures or graphical data per second. Libraries such as CUDA expose GPU features to programmers, which can exploit them to accelerate conventional computing; this approach is defined as General Purpose GPU (GPGPU) programming.

In video games the correctness of the image is not so fundamental, as some wrong pixel can not create great problems. Instead in the last two examples the accuracy of the image is fundamental. Every pixel must reflect reality and must be free of errors. Reliability becomes the main point, therefore it is mandatory to guarantee the proper operation of the GPGPUs in a defined period of time.

Background

Working with GPGPUs is not simple, as they present multiple streaming multiprocessors and work with parallelized threads. Therefore try to identify and correct an error could be an hard job.

One of the most used technique employed to improve reliability is the software redundancy. Software redundancy are the simplest techniques. The main idea is to replicate the code, in this way it is valid suppose that the replicated units are independent each others. Therefore if a fault occurs in a unit, the others are faults free, and so a valid result is always present. In case of more than two units a voter can be used. The voter selects the result produced by more units. Thinking that

just one of the unit produces a corrupted output and so a different result, the output will be the correct result. This technique is easiest, as all the codes can be used without any adapting. A complete different approach is Algorithm based fault tolerance - **ABFT**. This technique is based on the input encoding. The coding is used to check the result. The corrupted output is identified and corrected. ABFT is commonly used as error's detection in CPUs, but it is innovative in GPGPUs' environments.

Goal of thesis

Increase the reliability is a delicate job, as in most of the cases it implies a worsening in the execution time and/or in the occupied memory space. The presented work of thesis analyses the evolution of time and memory parameters respect to the improvement of the fault coverage.

Three different approaches to improve reliability were chosen and applied to matrix multiplication. The three techniques implemented are : Duplication with comparison - **DWC**, Triple modular redundancy - **TMR** (used to test redundancy's skills) and a modified version of the **ABFT**.

Due to some issues, codes were not tested on a real GPGPU but on **FlexGrip**, a soft GPGPU architecture which has been optimized for FPGA implementation. FlexGrip is based on the NVIDIA G80 architecture and it is implemented in VHDL. The advantage of using this FPGA-based is the possibility to customize at multiple level the hardware.

Case of study

Codes were written in **CUDA** - Compute Unified Device Architecture - an high level language able to manage the execution of multiple threads at the same time. CUDA uses particular functions called, **Kernel-call**, which pass the execution control to the GPGPU. Some test were performed on VisualStudio-2015 to check the correct behaviour of the code without any faults in a real GPGPUs.

In a real GPGPU the high-level code is transformed in an intermediate pseudo-assembly language, the Parallel Thread Execution - **PTX**. Virtual registers and pseudo-instructions mapped by PTX are transformed by binary language. The virtualization of register could create some issues depending on the system used to compile the code. Each codes were compiled in Linux-system and adapted for Windows-system. The binary languages obtained during the compilation are used as instructions to feed the FlexGrip. Through Modelsim, a VHDL-simulator, the three implementations are tested at low-level. At each simulation a **Golden Memory** is obtained, that is the ideal output without any errors.

In order to check the behaviour of the techniques an injection faults is required, which simulate the codes in presence of a fault. For each code was collected each

bit in which a **stuck-at** can occur. Therefore a simulation was performed for each position taken into account. At the end of the injection fault simulation a **Corrupted Memory** is obtained, which is a result with errors caused by the fault. To check how much the Corrupted is different from the Golden memory, a comparison is performed. Faults can cause three kind of errors, the thesis is focused on **SDC - Silent Data Corruption**. The presence of an SDC means that the simulation reaches the end but with a mismatch in the comparison.

The ABFT implementation is different from the others as it requires a triple kernel calls. Due to this difference simulation and also the fault injection needs some variations in the procedure.

Conclusion

Results demonstrates that the ABFT is not the best implementation, moreover it requires more time and space than the others. Taking care that three kernels are involved in the injection and therefore the starting matrices can be faulted and therefore the result, the partial fault coverage is not the worst. The goodness of this technique is due to the fact that the error can be detected and localized, so corrected. It is reasonable thinking that with some adjustment in the PTX-code, and so protecting the some registers, it could be improved. As expected, considering software redundancy technique, TMR is better than DWC, and also better that ABFT. Obviously TMR needs three times the parameters required by the DWC. To be faster, the DWC was modified, unfortunately the Nvidia compiler try to optimize the binary code. The optimization eliminates the voter presented in the code, making the code worse of the matrix multiplication without any improving technique. Some future improvements could be done. As example rewriting the code in "PTX-language" avoiding the elimination of the voter is one solution, or trying as for the ABFT to protect delicate registers, maybe using more of them avoiding to employ the same register for different purpose.

Chapter 2 overviews the internal structure of a Graphics Processing Unit, highlighting the difference with respect to CPUs and CUDA, the programming language that exposes the features to the programmer. Moreover, Chapter 2 introduces FlexGrip, the open-source model onto used for the present research.

Chapter 2

Graphics Processing Units

The present chapter is intended as a brief introduction to the architecture and operation of a Graphics Processing Unit (GPU).

A general-purpose GPU (GPGPU) is a graphics processing unit (GPU) performing non-specialized calculations, whereas such kind of device would be traditionally employed to perform image processing. The peculiarity of these devices is the ability to efficiently scale highly parallel workloads, which makes them preferable to CPUs for such tasks.

2.1 CPU vs GPU

In order to have a better comprehension of the future paragraphs, this section will outline the differences between a Central Processing Unit (CPU) and a Graphics Processing Unit (GPU).

GPUs and CPUs are devices with very different architectures, the former is designed for running a large number of same operation, whereas the latter is designed to simultaneously execute at most a handful of potentially complex tasks.

CPUs traditionally have a single core, with modern devices featuring two or four cores typically. On the other hand, the FlexGrip Fermi architecture features 16 Streaming Multiprocessors (SMs) executing code in parallel, and therefore producing 32 pieces of data per SM.

Furthermore, the number of registers in a CPU to be used during the execution of a given task is relatively small, requiring the CPU to perform what's known as a *context switch*. When a subroutine is called, the entire set of registers must be saved and later restored. When tasked with several simple operations, CPUs switch between them, spending most of the time performing the context switches than actually performing the execution. GPUs instead have multiple banks of registers, and that coupled with increased number of "cores" removes the context switch overhead.

2.2 GPU architecture

The follow section is meant to detail a GPGPU as generally as possible, taking as reference the NVIDIA Tesla microarchitecture. Programmable using CUDA or OpenCL APIs, this architecture unifies both vertex and pixel processors, both of which will be covered in the present section, in the graphics pipeline.

2.2.1 Graphics Pipeline

A graphics pipeline is a sequence of steps performed to map a perspective of a 3D environment onto a 2D screen. The main steps are two, represented in the image below.

Geometry

Geometry is defined as what is meant to be displayed on the screen. Points and shapes can be used for this purpose and they are called geometric primitives. Such objects are then converted into points or vertices and the result is stored in memory. One should note that vertices carry colour information, along with spatial references.

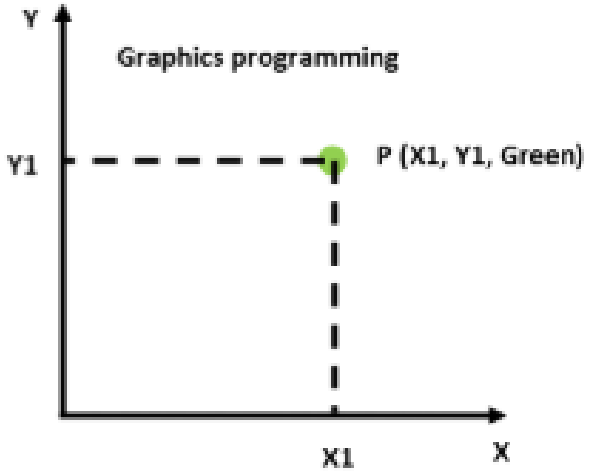


Figure 2.1: Information of a vertex in a 2D space.

Vertex shaders process the geometry further, mapping the position of each vertex in the 3D virtual space to the 2D coordinate at which it appears on the screen as well as the illumination of that point. The output of such program is fed to the rasterizer.[1]

Rasterisation

The final step preceding the pixel pipeline, a rasterizer maps the infinitely precise vectorial 2D representation to a finite amount of pixels. As can be seen in Figure 2.2, a triangular shape is broken down into the fragments composing it, coloured in grey.

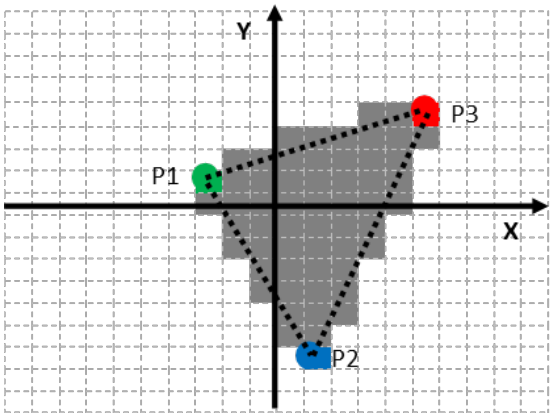


Figure 2.2: The rasterization process

An overview of the pipeline can be seen in figure 2.3. [2]

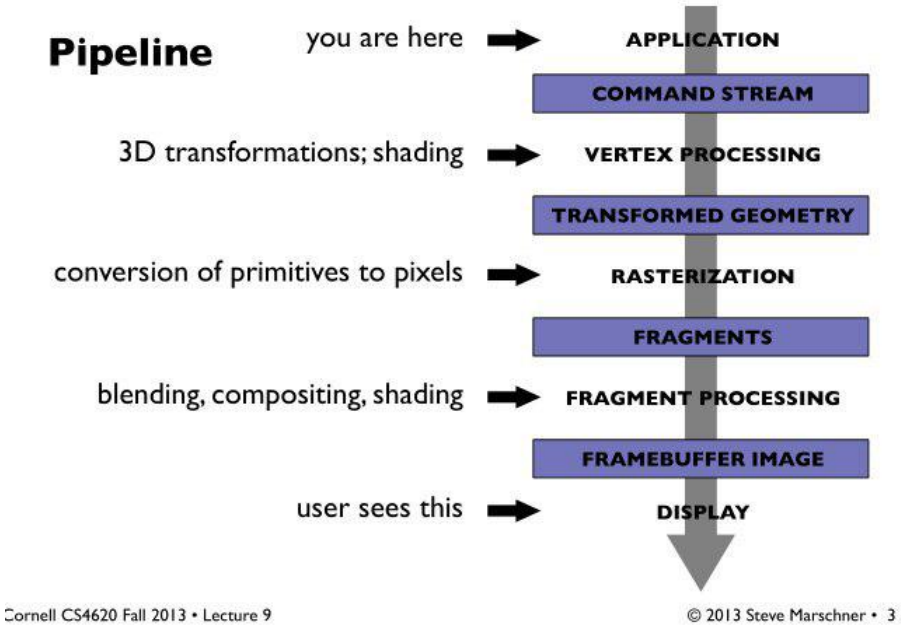


Figure 2.3: The graphics pipeline

2.2.2 Tesla microarchitecture

The present microarchitecture was employed in NVIDIA's GeForce 8800 Series, specifically in the 8800 GTX graphics card, which will be taken as reference in the following analysis. This device features eight independent units named Texture Processing Cluster (TPCs), each composed by 16 streaming multiprocessors (SMs) for a total of 128 SMs. [12]

Moreover, a geometry controller, an SM controller (SMC), two streaming multiprocessors (SMs), a texture unit and two blocks of cache memory are built into each TPC.

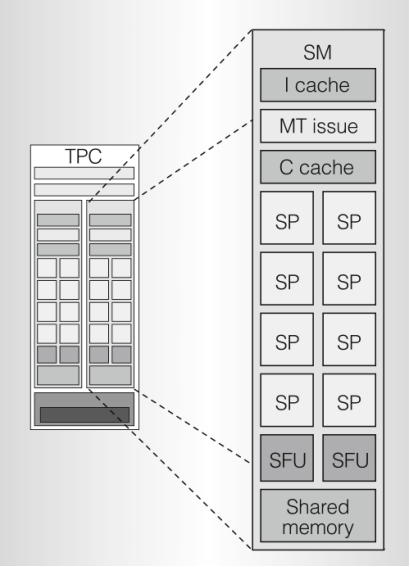


Figure 2.4: TPC block diagram

Streaming Multiprocessors are the units executing vertex, geometry, and pixel-fragment shader programs and parallel computing algorithms. As Figure 2.4 shows, each SM inside a TPC includes two Special-Function Units (SFUs), containing floating-point multipliers.

An SM concurrently executes different threads, each having their own thread execution state, enabling for multithreading.

Stream Processors (SP) are the primary thread processors, performing all basic operations, such as addition, multiplication and multiply-and-accumulate, both in integer and floating point formats.

Single-Instruction Multiple-Thread (SIMT) is a processor architecture introduced by NVIDIA in the Tesla architecture. This unit is used to manage and execute all the threads. The threads are divided in groups of 32 elements, called *warps*. Each element inside of the same warp is scheduled to be executed in parallel.

Threads belonging to the same warp are started together but are free to branch and execute independently. The SIMT selects a warp to execute an instruction and issues the next one to another warp. SPs are mapped by the SM to manage a warp.

Memory

A Streaming Multiprocessor implements memory load/store instructions to support C/C++-like languages. For this reason to compute load/store instructions, the access to three different memory is allowed. In the following they are presented:

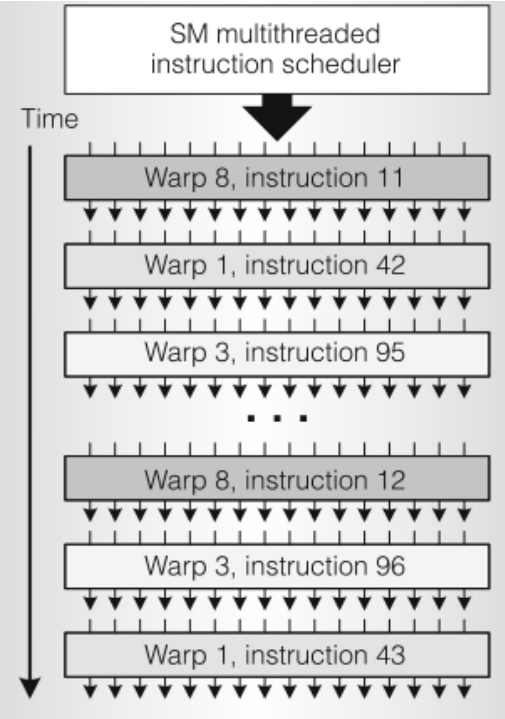


Figure 2.5: WARP execution diagram

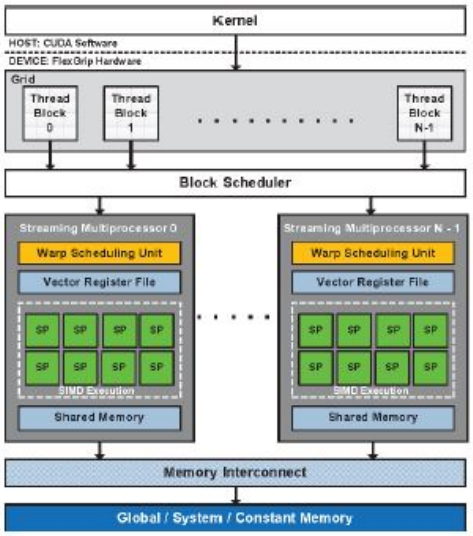


Figure 2.6: GPGPU high level schematic

- **Local memory** is owned by each thread, and it's a private space, where the others threads can't access. This particular memory is implemented in external DRAM.
- **Shared memory** is the second type of memory. This is a space where each thread of the same SM can make the access. Shared is useful to share data and create cooperation.
- **Global memory** can be accessed by all the threads, even if they are not in the same block.

In order to use Global and Shared memory some synchronization instructions are used to avoid issues.

2.3 CUDA

Compute Unified Device Architecture(CUDA) is an API designed by NVIDIA to augment C and C++ languages, by means of CUDA-accelerated libraries or compiler directives. CUDA allows for direct access to the GPU's parallel computational elements. Therefore, a programmer can write a single instance of a CUDA program and scale it to multiple threads simultaneously.[16],[7].

A CUDA program may be split in two sections, one executed by the CPU in the system memory (the *Host*) and the other by the GPU (the *Device*). The main steps executed in a CUDA code can be summarized as:

1. Declare and allocate host and device memory.
2. Initialize host data.
3. Transfer data from the host to the device.
4. Execute one or more compute kernels.
5. Transfer results from the device to the host.

A kernel is a particular function that when called are executed several times in parallel by threads on the device. A kernel function is formally defined by `""__global__""` and describes the operations to perform. Such function is called by `<<<.....>>>`.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Figure 2.7: Example of a kernel-function

2.3.1 Threads, Blocks and Grid

This section will discuss and define the concept of thread. A thread is the fundamental building block of parallel programming, meaning that multiple instances of one may be executed simultaneously, each with their own index. Threads exist in all the three dimensions, therefore have an index per each dimension.

- ThreadIdx.x
- ThreadIdx.y
- ThreadIdx.z

Threads are grouped in blocks, also existing in the three dimensions, in turn are indexed in the three coordinates.

- BlockIdx.x
- BlockIdx.y
- BlockIdx.z

When a kernel is called, CUDA allows to decide how many blocks instantiate and how many threads per block.

```
<<<N_blocks,N_thread_x_block>>>
```

The number of threads per block is defined by “blockDim.x/y/z”, feeding the three dimensions with the elements. A block cannot contain more than 1024 threads. Multiple blocks may be instantiated, determining their position along each axis with ”GridDim.x/y/z”. Different blocks in the same grid can not communicate with each other.[**altro2**]

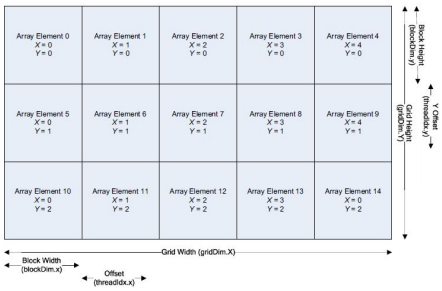


Figure 2.8: Block structure

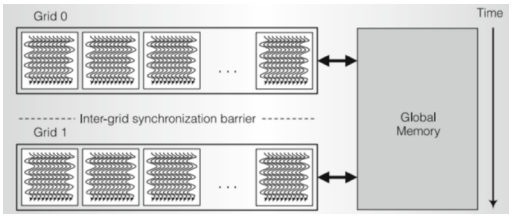
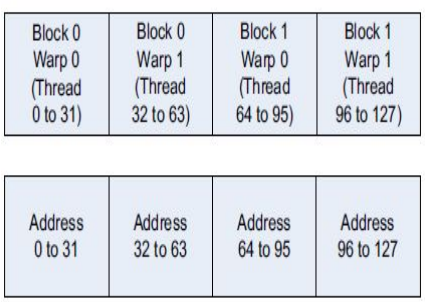


Figure 2.9: Grid organization

Each kernel has its own grid with its own specifications, meaning that kernels can not be executed concurrently, but only one at a time.

A set of blocks is assigned to a SM and each block is split in warps, therefore the number of threads in each block must be a multiple of 32. Each SM executes a pool of warps, with a separate instruction pointer for each warp.



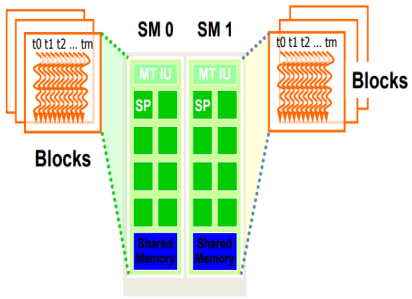


Figure 2.10: Blocks’ organization

During the years different versions of CUDA have been realized and updated. As you will see in the next section, the thesis work was done not on a real GPGPU, but on FlexGrip.

This "device" simulates the behaviour of a real GPGPU, Nvidia G80. The version of CUDA that works on the architecture of the G80 is `version 1.0`.

In this first generation CUDA compute capability 1.x (`sm_1.0`) device, some instructions and operations are not supported. For example, donormal numbers are unsupported, the precisions of the division and square root operations are scarce. Moreover this kind of architecture - `FERMI` -, does not support atomic operations in any of the three memory types, and does not allow the use of the command `__synchronize()`. This command is used to create a synchronization between threads, which try to work on the same space in global or shared memory, avoiding race condition.

2.4 FlexGrip

The present section describes the soft GPGPU architecture on which the work of thesis was performed.

FlexGrip is a soft GPGPU architecture optimized for FPGA implementations. It is based on the NVIDIA G80 architecture, and implemented in VHDL. The benefits of using such soft processor is the ability to specify the functionality in a high-level language (e.g. C or CUDA) and the flexibility to tailor such functionality to the FPGA target. FlexGrip provides the test engineer with the ability of specifying the amount of parallelism at multiple levels, changing a file named`pickBench.vhd`[4].

FlexGrip is able to execute both Integer and Floating Point instructions on 16 or 32 bits and perform arithmetic and logic operations, move entries between the different levels of memory and other instructions. [15]

2.4.1 Architecture and Functionality

Figure 2.11 depicts the architecture of FlexGrip, a briefly description of the main concept is presented. The warp unit coordinates the instructions in the pipeline.

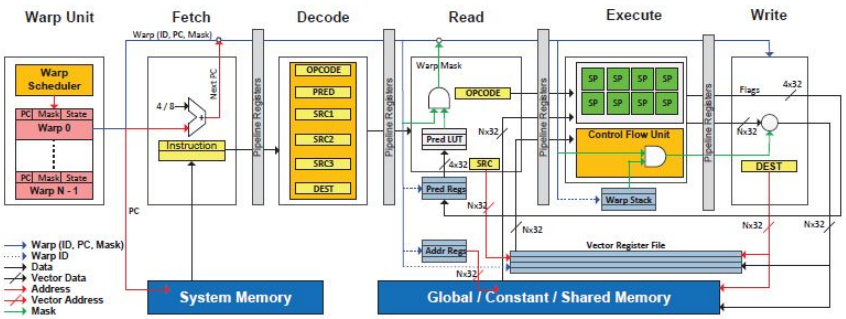


Figure 2.11: Architecture of FlexGrip

The SM pipeline architecture is composed by five stages: Fetch, Decode, Read, Execute and Write. Since the SIMT model is employed, an instruction is fetched and mapped on multiple scalar processors simultaneously. Blocks of threads are scheduled in a Round Robin fashion.

Each warp has its own program counter, therefore can follow its own conditional path.

One of the abilities of FlexGrip is managing the conditional branch that could cause a divergence, a "barrier" that only some threads can pass through. Should such event occur, execution for some SPs continues (i.e. branch taken), when the divergence is resolved, the execution switches to the other SPs (i.e. branch not taken).

Software Flow

Being FlexGrip a hardware model, it must be provided with binary code to execute. CUDA code is compiled with `nvcc`, the NVIDIA compiler that converts the high-level C-like code into an intermediate pseudo-assembly language, named `Parallel Thread Execution (PTX)`. [15]

Optionally, the programmer may choose to write programs in PTX directly. The program compiling PTX virtual registers and pseudo-instructions to binary language is copyrighted by NVIDIA, hence its source code is not available to the public. The virtualization of the registers, meaning their position in global memory, changes according on the system used. In fact Windows and Linux allocate registers in different methods.

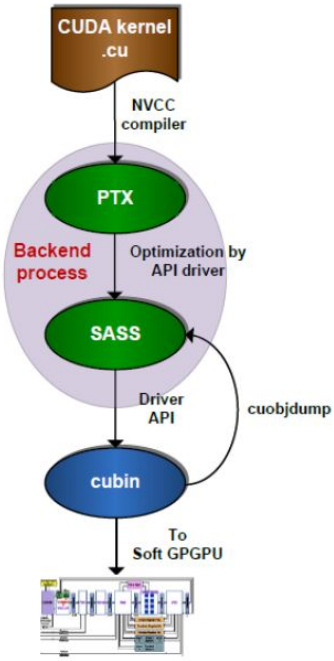


Figure 2.12: Software Flow

This process is performed by the CUDA driver API, that using `cuobjdump` tool, realizes a human-readable hardware language, named `Source and ASSembly code (SASS)`. This represents the actual code executed on NVIDIA devices.

The described process is shown in Figure 2.12. Note that during the creation of SASS files, the compiler may perform some optimizations, which may inadvertently alter the program structure and cause unexpected behaviours. This topic is discussed further in Chapter 3

Using the software Nsight Eclipse Edition the process above description is performed, moreover a file called `TP_instructions` is created. This file feeds the FlexGrip hardware and it is used as set of instructions to be performed.

Chapter 3

Reliability

This section will provide a brief introduction to software redundancy. Let us start with the definitions of the core concepts of such topic.

A generic device can be seen as a system communicating and interacting with other systems. A defect is a distortion of the physical structure at the hardware level. A fault is defined as an abstraction to model and represent defects [5]. Faults can be categorized as:

- **Permanent** - it remains present until the system is repaired or replaced. (i.e **stuck-at**)
- **Intermittent** - appears repeatedly in regular and non-regular time intervals.
- **Transient** - occurs once and disappears.

When a fault occurs in a system, it may deviate its behaviour from its specifications, manifesting in what's defined a `failure`. A failure implies that an external state differs from the expected state. This variation is called `error`.

Reliability is the probability that a system provides its correct service during a specified period of time. [5] Fault prevention and fault tolerance are approaches aimed at increasing reliability, that is preventing faults before they occur. Fault tolerance focuses on `error detection` and `error recovery`.

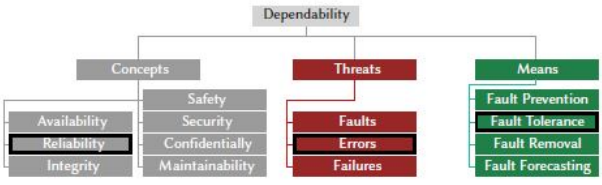


Figure 3.1: Scheme of concepts

The goal of this thesis is to improve the reliability of GPU matrix operations. To achieve this, different approaches were followed.

3.1 Software Redundancy

Software redundancy is the first technique that was experimented with. Such approach may be implemented either in hardware or software, by means of replication of components in the hardware domain, or functionalities in software. While the latter technique does not require any modification to the physical system, it is as susceptible to failures as an unprotected system.

The assumption that programming faults are fully independent and therefore will unlikely fail on the same input is at the base of these techniques. While the approaches considered are based on code redundancy, other forms of redundancy may be employed, such as the environment or data.

Redundancy may be *deliberate* or *intrinsic*, the difference being the developer's intention. [13]

3.1.1 Deliberate Redundancy

This case implies the intention in the engineer to introduce redundancy in the system for a specific purpose. All the techniques that replicate the design process to produce redundant functionalities are part of this category.

As an example, let us consider a program performing a task. For this purpose, one may employ N different algorithms, for N-way redundancy. This way, that is using several versions separately, it is assumed that they will be based on entirely different designs, and thus not susceptible to correlated failures. [11] Implementing this procedure is significantly more expensive, and for this reason it is used exclusively in environments where any failure is not acceptable, such as in the aerospace industry.

3.1.2 Intrinsic Redundancy

On the other hand, intrinsic redundancy is not deliberate, but rather is a natural consequence of the aspects of the design combining. Several executions of a program are considered redundant when they have the same observable behaviour, that is the user can not tell the two outputs apart.[6]

Several approaches exploiting software redundancy techniques exist. This work of thesis considered Duplication With Comparison (**DWC**) and Triple Modular Redundancy(**TMR**) both of which will be explained in detail in the following sections.

3.2 Duplication With Comparison (DWC)

Duplication With Comparison is the easiest software redundancy technique to implement, simply consisting of performing the same operation or set of operations

twice. In the present case study, matrix multiplication instructions are doubled, and the computed results are compared at the end. If a match occurs, the now confirmed correct result is stored, else both the two results are discarded.

The advantage of this technique is its simplicity, as it can be applied to any kind of program without any major change in its structure. On the other hand, large pieces of code may not fit on the available memory if duplicated.

3.3 Triple Modular Redundancy (TMR)

Triple Modular Redundancy is a technique used to improve systems' reliability using software redundancy approach.

The use of redundancy is not meant to be a replacement, but as a complement to the two cardinal principles of reliable design [14], [9]:

1. Use the most reliable components.

2. Reduce complexity as much as possible.

Figure 3.2 shows a scheme illustrating the concept of TMR, which will be used to explain this approach.

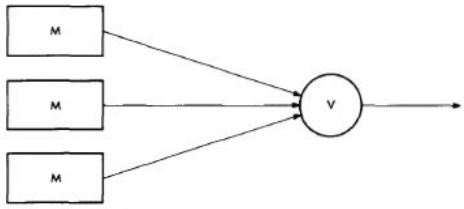


Figure 3.2: Scheme of TMR

The main idea is to triplicate the units that could be affected by faults. Supposing that an error is produced by one of the units, while the other two will produce a correct one. A voter decides what the correct result is and outputs it, effectively masking the error.

As can be seen in Figure 3.2, the box called `M` - the *module* - is replicated three times. Each of them are fed with the same set of inputs, and will produce a result.[10],[3]

All three results are in fed to entity `V`, defined as `majority organ` or `majority voter`. The voter takes as inputs the three preliminary outputs and, using a majority approach, produces the "true" output. Since the number of sources is odd, in the event of single fault occurrence there must be an unambiguous majority opinion.

This approach works if none of the units are affected by faults, or if just one of them is affected, meaning that it is assumed that the failures on the three modules are uncorrelated.

Voter ideality is another important assumption performed, but in fact if this module fails, the output will always be corrupt.

Some changes can be done to the scheme to improve the system, and therefore take into account the possibility that the voter may also fail.

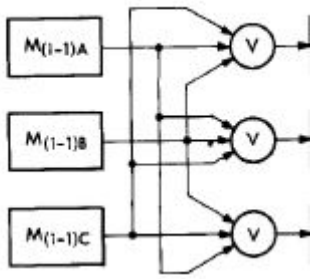


Figure 3.3: Scheme of improved-TMR

As Figure 3.3 shows, the voter can be instantiated multiple times. Each of the instances receives as inputs the outputs of each unit. This new scheme takes into account the possibility that the voter may fail. Obviously, this second TMR iteration is more expensive than the first, as in this case not only the modules, but also the voters are replicated.

This last approach can be employed when small matrices are considered. When the size grows beyond 256x256, a significant performance degradation occurs, not to mention an increase in the total power consumption of the system. These drawbacks are compensated by the generality of this approach, as the modules may implement an arbitrary algorithm without the need for re-engineering the system structure. [14]

```
1  output_c[row * B_w + col] =
2    (temp1 & temp2) | (temp1 & temp3) | (temp2 & temp3);
```

Figure 3.4: Example of the used voter

The voter used in the present work of thesis is depicted in Figure 3.4. As one can see, each element is compared with the other two, and only one output is stored, (a matrix multiplication in this case, that will be discussed later), as the majority dictates.

3.4 Algorithm based fault tolerance - ABFT

This approach is tolerant to fail-continue failures: in this operation mode, applications are modified to operate on encoded data, which allows for error detection and correction. This kind of approach does not employ any redundancy, but instead applies algorithm to linear algebraic computations, with the advantage of having a very low overhead.

The use of this approach is innovative in a GPGPU's environment, as traditionally it was reserved to CPUs. [14], [9]

As previously stated, ABFT requires the inputs to be encoded using a procedure that enables the test engineer to verify that the outputs were computed correctly exploting the mathematical properties of the encoding. If a fault were to occur, and therefore the output be incorrectly produced, the failure would be caught and masked before being propagated to memory.

Specifically, in matrix multiplications the input matrices A and B are encoded before being processed. As Figure 3.5 shows, an extra row A_c and an extra column B_r are merged with A and B respectively. These additional elements are defined as *check-sums vectors*. The $\mathtt{j}_{th}$ element of the added row of A contains the sum of all the elements in the $\mathtt{j}_{th}$ column of A. Similarly, the $\mathtt{i}_{th}$ element of the extra part of B is used to store the sum of all the elements in the $\mathtt{i}_{th}$ row of B.

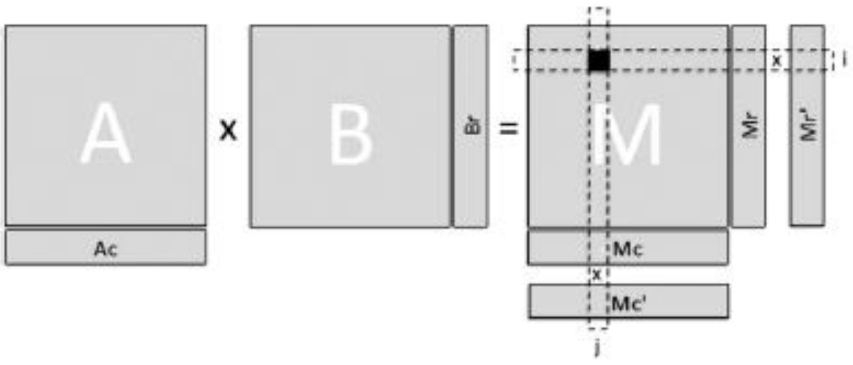


Figure 3.5: ABFT overview

Arithmetic operations are then performed normally on the operands, and if no errors occurred the resulting matrix would also be encoded. Much like the modified inputs, the output matrix M would be larger by an entry of each dimension; the additional row and column are defined as M_c and M_r.

Additionally, the element-wise summations of the n rows and columns of M is computed and stored in vectors M'_c and M'_c respectively.

$$\sum_{i=1}^{n} m_c^i = M_c \qquad\qquad \sum_{j=1}^{n} m_r^j = M_r$$

Comparing these two checksum vectors with a known reference can detect an error occurrence and correct it.

$$A \times B = M$$

However, such simple encoding can't correct more than one fault at a time, and is therefore not as efficient as more advanced techniques. If a fault is not correctable, the system must perform the entire matrix multiplication again, reducing the overall performance.

3.4.1 ExtABFT

Since the occurrence of multiple transient faults is a real possibility for some applications, more advanced techniques are employed to avoid repeating operations more often.

Overview

ExtABFT is an extension of the ABFT approach, computing multiple checksums providing additional coverage to certain matrix regions. The cost for such operations progressively grows as more and more corrections are required higher [4]. This enhanced version is able to detect multiple errors and correct them, the recomputation is done only if strictly necessary.

The principle of operations does not differ dramatically from plain ABFT, increasing the input size by one dimension each with the additional row vector *faulty_col* and column vector *faulty_row*. During checksum calculations, if an error is found a counter is incremented, and then used as pointer in the *faulty vector*.

After all the errors are identified, they must be corrected. The faulty vectors are used as indexes and corrected using M'_r or M'_cas necessary. Only the corrupt cell is accessed, and all the other cells in that row (or column) are subtracted from it. This system can correct errors occurring not only in a single cell location, but also any other in the same row or column.

For more sparse error correction, one must employ even more refined solutions, which will not be covered in the present thesis.

Modifications

The ExtABFT approach followed underwent some tweaking, specifically The code used is able to identify only one error, but can be easily modified to recognize multiple errors.

Similarly to the aforementioned version, during the comparison *faulty_col* and *faulty_row* are used to take trace of the error. A counter also in this case is present and used as index in the two vectors. Differently from vanilla ExtABFT, the index of the corrupted cell is not overwritten, instead just setting a cell.

During the correction phase, the two faulty vectors are visited looking for a high bit. The index in *faulty_col* is the x-coordinate and the index in *faulty_row* is the y-coordinate of the error in M. As in the original version M_c or M_r are used to fix the cell.

Chapter 4

Case Study

The present chapter will outline the experimental work performed during the thesis, whose purpose is finding ways of improving a system's reliability using software redundancy.

4.1 Software

Initially, the algorithms implemented in CUDA were Duplication With Comparison (DWC), Triple Modular Redundancy (TMR) and Algorithm-Based Fault Tolerance (ABFT), using in Visual Studio IDE. During this phase different strategies were analysed. Each code was written considering the three types of memory and matrices of different size.

Varying the dimension of a matrix also changes the number of threads per kernel, as showed in Figure 4.1.

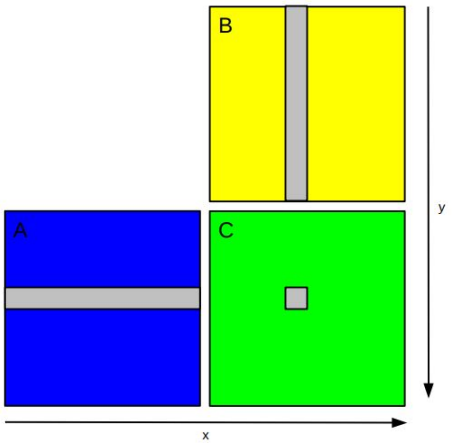


Figure 4.1: How threads manage matrices

When a multiplication between two matrices is performed, each cell of the resulting C-matrix is computed by a single thread. The Index-x of this thread represents the row of the first matrix (A), and the Index-y the column of the second matrix (B) involved in the multiplication.

Therefore, as each thread manages a single cell of the C-matrix, there are many threads as the number of cells in the resulted matrix. Obviously, increasing the size of the input matrices, also the size of the resulted one is increased, and so the number of threads involved in the multiplication.

In order to consider the worst case global memory was chosen, as this is the case that requires more time. Summarizing:

- DWC requires six accesses to the memory, two load instructions for the matrices A, two load instructions for the matrices B and two store instructions for c matrices.
- TMR requires nine accesses. Six load instructions for inputs matrices and three store instructions for the outputs.
- ABFT requires just three accesses to the memory, two for inputs and one for the store.

The approaches above share some similarities, as some functions were used for all of them. As already said, the `main` is the part of the code executed by the Device or CPU. The pointers to the memory of the input and output matrices are created in this space. The main is also in charge to feed with random values the input matrices. As Figure 4.2 shows, the dimension of the C matrix is given taking care of A's heigh and B's width.

```
1     int a[arraySize] = { 0 };
2     int b[arraySize] = { 0 };
3
4     for (int j = 0; j < arraySize; j++)
5     {
6     a[j] = 1;
7     b[j] = 10;
8     }
```

Figure 4.2: Matrices creation

Pointers, (e.g the matrices), are passed to the the host or GPU, using `addWithCuda` function. All the operations computed by the host are in this function.

At the beginning some checks to verify the correctness of the communication between GPU and CPU are done. After this step, the host allocates some space in its global memory for the matrices. Input matrices were already created by the device, therefore they are just copied from the device's memory to the host's global memory.

As already said, multiple multiplications were performed changing the size of the matrices involved. At every time the same kernel with different description was called.

Figure 4.3 shows the calling of function `dimension()`. The size of matrices may be significant, in turn the number of threads managing them; a block may be composed by only 1024 elements.

```
dimension(A_heigh128, B_width128, GX, GY, BX, BY);
printf("GX=%d, GY=%d, BX=%d, BY=%d \n", GX, GY, BX, BY);
dim3 block_x128(BX, BY, 1);    // Threads per block
dim3 grid_x128(GX, GY, 1);     // Blocks per grid
MatrixMul <<< grid_x128, block_x128 >>> (dev_a, dev_a2, dev_b,
dev_b2, dev_128c, dev_128c2, A_heigh128, B_width128, A_width);

dimension(A_heigh256, B_width256, GX, GY, BX, BY);
printf("GX=%d, GY=%d, BX=%d, BY=%d \n", GX, GY, BX, BY);
dim3 block_x256(BX, BY, 1);      // Threads per block
dim3 grid_x256(GX, GY, 1);       // Blocks per grid
MatrixMul <<< grid_x256, block_x256 >>> (dev_a, dev_a2, dev_b,
dev_b2, dev_256c, dev_256c2, A_heigh256, B_width256, A_width);
```

Figure 4.3: Same kernel, different dimension

"Dimension" describes the number of blocks used by a specific kernel. This function takes as input the dimension of A and B matrices and derives the dimension of the C-matrix. If the size is bigger than 1024, it divides the threads in smaller blocks. As the division between the number of elements and 1024 could be a non-integer number, `Rounding_up()` function is used. The purpose of this new function is just to guarantee that the number of blocks involved is an integer number.

```
int rounding_up(int x, int y)
{
int result = 0;
if (x != 0)
{
result = ((x - 1) / y) + 1;
}

else
{
result = 0;
}
return result;
}
```

Figure 4.4: rounding_up() function

```
1   void dimension(int A_h,int B_w,int& G_X,int& G_Y,int& B_X,int& B_Y)
2   {
3   if (B_w*A_h > 1024)
4   {
5
6   int numBlocks = rounding_up(A_h*B_w, 1024);
7   int numThreadsPBlock = rounding_up(A_h*B_w, numBlocks);
8   int x = rounding_up(A_h, B_w);
9
10  int w2 = rounding_up(numThreadsPBlock, x);
11  int w = rounding_up(sqrt(w2), 1) + 1;
12  int h = w * x;
13
14  int IBx = rounding_up((B_w - h - 1), h) + 1;
15  int IBy = rounding_up((A_h - w - 1), w) + 1;
16
17  B_X = w;
18  B_Y = h;
19
20  G_Y = IBy;
21  G_X = IBx;
22  }
23
24  else
25  {
26  B_X = B_w;
27  B_Y = A_h;
28  G_X = 1;
29  G_Y = 1;
30  }
31  }
```

Figure 4.5: dimension() function

One may instantiate more threads than necessary, but to avoid issues with the extra elements, some limits were put in place, in the form of `IF-Statements`. The purpose of "IF" is to check if the thread's index is bigger, and so identified as extra element, than the matrix dimension. As the matrix has two dimensions "IF-Statements" checks two limits, as follows:

```
1   if (row < A_h && col < B_w)
```

In the next section the peculiarity of "ABFT" code and "DWC" will be taken into account.

4.1.1 DWC

In the previous section was told that different approaches changing the type of memory were performed. At the end all the codes were implemented using global memory.

Respect to the other codes, two version of the DWC were tested. One implemented using global memory, as for the other cases, and one using local memory.

In the first version the result for the two versions were computed and stored respectively in "output_c" and "output_c2". Then the two results were compared. The comparison were performed between two global memory locations, that means load from the memory and comparing. Therefore these steps are required:

1. Two store instructions for "output_c" and "output_c2".
2. Two load instructions to takes values to compare.
3. Comparison.
4. Other steps in case of a discrepancy between the two values.

Global memory is the slower memory of the GPU, and the multiple accesses listed above make the process slow.

The purpose of the second version was to optimize and try to make the process faster. To reach this goal, local memory was used.

In this second version, called **DWC2**, only three accesses to global memory was performed. Two reads for inputs and one write for the output. In contrast to the "`global-DWC`", DWC2 computes the results in temporary variables. Temporary variables are variable stored in local memory, therefore existing only inside that thread, and not accessible from others.

The comparison is made between two local variables. If they are equal, only the result of the first multiplication is stored in the global memory. Therefore this step does not require to load the results form global memory. Referring to the list of operations listed above step 2 is not any more required.

Using this second version a step was skipped, and the operations were made faster as performed between local memory.

4.1.2 ABFT code

Generalities

The functionalities of the ABFT was already explained, this subsection's purpose is to give an overview of how the code working.

Differently from the other codes, the ABFT requires three different kernels. The first kernel is used to create the last row (i.e `check-columns`) of matrix A. The second kernel is similar to the fist one, its purpose is to calculate the last column (i.e `check-rows`) of matrix B. In the last kernel the matrix multiplication is computed using the outputs of the first two kernels. In the other codes, after a kernel finished, the memory allocated for matrices in GPU were cleaned. In this case, instead, memory is conserved and passed to the third kernel.

- Check-Sum A = first kernel.
- Check-Sum B = second kernel.
- MatrixMul = third kernel.

As for the other codes, also in this case multiple matrix's dimensions were implemented. Due to divergence's problems, the size is smaller than the others case. The maximum size used is 32x32. This dimension is the maximum reached because it is the warp's size, therefore all the threads execute the same instructions avoiding divergence.

Every time the size is changing, each one of the three kernel must be called again. In this section a call to the ABFT means a call to three different kernels.

```
1  cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int),
2  cudaMemcpyHostToDevice);
3  if (cudaStatus != cudaSuccess) {
4  fprintf(stderr, "cudaMemcpy failed!");
5  goto Error;
6  }
7  cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int),
8  cudaMemcpyHostToDevice);
9  if (cudaStatus != cudaSuccess) {
10 fprintf(stderr, "cudaMemcpy failed!");
11 goto Error;
12 }
13 dimension(A_width, 1, GX, GY, BX, BY);
14
15 dim3 block_a(BY, BX);
16 dim3 grid_a(GX, GY);
17 //sistema poi 1 per blocchi più grossi
18 checkSumA <<< grid_a, block_a >>> (dev_a, A_heigh, A_width);
19
20 dimension(1, B_heigh, GX, GY, BX, BY);
21 dim3 block_b(BY, BX);
```

```
22  dim3 grid_b(GX, GY);
23  checkSumB <<< grid_b, block_b >>> (dev_b, B_heigh, B_width);
24
25  dimension(A_heigh + 1, B_width + 1, GX, GY, BX, BY);
26  dim3 block_x(BX, BY, 1);       // Threads per block
27  dim3 grid_x(GX, GY, 1);         // Blocks per grid
28  MatrixMul <<< grid_x, block_x >>> (dev_Faulty_row, dev_Faulty_col,
29  dev_Mcc, dev_Mrc, dev_a, dev_b, dev_32c, A_heigh +1, B_width +1, A_width);
```

Listing 4.1: Steps to compute matrix multiplication using ABFT

As the figure 4.1 shown, an new function is used. In the others codes matrices A and B were every time equal. As they were defined as a long vector, changing the dimension means using just part of the vector. In this case `Check-sum A` and `Check-sum B` add elements to the original matrices. For that reason it is impossible define just a long vector. `matrix_a_b()` solves this issue.

```
1   void matrix_a_b(int *a, int*b, int ha, int wa, int hb, int wb)
2   {
3   int value_random=0;
4   for (int i = 0; i < 400; i++)
5   {  //clear matrix
6   a[i] = 0;
7   b[i] = 0;
8   }
9   //feed A
10  for (int j = 0; j < (ha*wa); j++)
11  {
12  a[j] = value_random;
13  value_random++;
14  }
15  //feed B
16  int value_random2 = 0;
17
18  for (int j = 0; j < hb; j++)
19  {
20  for (int p = 0; p < wb + 1; p++)
21  {
22  if (p == wb)
23  b[value] = 0;
24
25  else
26  b[value] = value_random2;
27
28  value_random2++;
29  }
30  }
31  }
```

Listing 4.2: Input matrices definition

Figure 4.2 explains how this function works. At every time this function is used, the memory allocated in the device is cleaned. So A and B are fed taking in account that B must have the last column empty, and A must have the last row empty.

Seeing 4.1, it is clear that after `matrix_a_b()` is performed, the new input matrices must be stored in the host's memory before ABFT is called.

`MatrixMul`

Now `MatrixMul` kernel is presented. The functionalities of this kernel, can be distinguished in four phases.

1. Matrix Multiplication
2. Check-sum row and Comparison
3. Check-sum column and Comparison
4. Correction

The second and the third points are equal to what is performed by `Check-sum A` and `Check-sum B`, not considering the comparison stage explained later. Not all threads involved in the matrix multiplication are present in these phases. In fact for each point a divergence is created. Check-sum row is performed only by threads that have ThreadIdx.y equal to zero, in other words only threads that manage the first C-row. Similarly check-sum column is performed only by threads present in the first c-coulumn, (i.e ThreadIdx.x equal to zero).

When the comparison vector is computed, it is used to check if a discrepancy is present. If so an extra vector sets a flag considering the row or the column with a fail. If an error is present on a row, also a column presents an error. *faulty_row* and *faulty_col* are the vectors used to store the error's position.

The last step is the most complicated. To avoid race condition, only one thread is chosen to comparing the last column and the last row of C with the two extra vectors computed in steps two and three. In particular the thread in position (0,0) in the C-matrix.

A loop searches in *faulty_row* and *faulty_col* high flags and stores in two pointers the location of the flag. In this way the failed cell is pointed by two coordinates, and can be corrected.

it is important to notice that multiple errors can be managed only if one coordinates is in commune with the others errors.

4.2 Simulation

The presented codes were tested in a previous phase on Visual Studio, and than they were run on FlexGrip. In a first moment a real GPGPU was the target of this thesis but due to some inconveniences it was opted to a FPGA GPGPU-based.

Simulation was performed using Modelsim and Nsight Eclipse Edition. As already said FlexGrip is fully configurable, this is essential in fact codes requires different specifications.

Specifications are initialized using a file called `PickBench`. This file is written in VHDL hardware description. The number of threads, blocks and cores are here described. Furthermore this file has a section to identify variables and pointers to memory passed from the device to the host in the real GPGPU.

To create input matrices a python script was used. This file, called `memGen.py`, generates random integer values that are converted in hexadecimal. Each hexadecimal value is divided in four pieces, one under the other and saved in a file called `Global_mem`. This file has a particular extension type: Memory Initialization Files (**.mif**). Global_mem.mif represents the global memory used by the FlexGrip.

Cuda codes are transformed in machine language using Nsight Eclipse, as described in the previous chapter. In this steps two files are created `Instructions_TP.vhd` and `Configuration_TP.vhd`. The process to realize the instruction's file was already discussed in the CUDA's chapter. This section focuses on the problem that occurs during the creation of this file.

The environment used to create this file is important. In fact, as already said, some virtual registers are allocated to compute instructions. This step is common in all the operative systems, the difference is how they allocate them.

Linux is used to place registers in growing order in increasing addresses, but not using continuous address. Instead, Windows is used to allocate registers in an optimized way, trying to compress them. Then use continuous increasing addresses.

It is important to notice that the binary codes produces by one system are not equal to the other.

Unfortunately Nsight Eclipse Edition worked on Linux environment, instead the Modelsim simulation worked on Windows environment. Therefore most of the time the binary codes were manually changing. 4.6 shows the correction made to the addresses from Linux to Windows.

4.2.1 Simulation's Configuration

Codes were written using matrices with different sizes. Simulations were done using only one size. In particular matrices had eight elements on the x-axis and eight elements on the y-axis. Therefore each call kernel required one block, and 6 -threads.

PickBench files are pretty much equal for all the codes, only the constant part is different. In this part are written input addresses, then it is different depending on how many inputs and outputs are used (I.e. how many parameters are placed in the kernel definition.).

Linux	Windows
0x04	0x04
0x06	0x05
0x08	0x06
0x0a	0x07
0x0c	0x08
0x0e	0x09
0x10	0x0a
0x12	0x0b
0x13	0x0c
0x14	0x0d

Figure 4.6: Memory offsets in Linux vs. Windows

Configuration file is common for all the codes. Obviously instruction file is different for all the codes.

At the end of Modelsim simulation a file is generated. `rdata_mem.txt` is the resulted of the simulation. It represents the global memory, in the addresses decided in the pickBench phse, can be seen input and output matrices. The actual operation of the codes can be checked in the resulted file or looking the waves generated by the program.

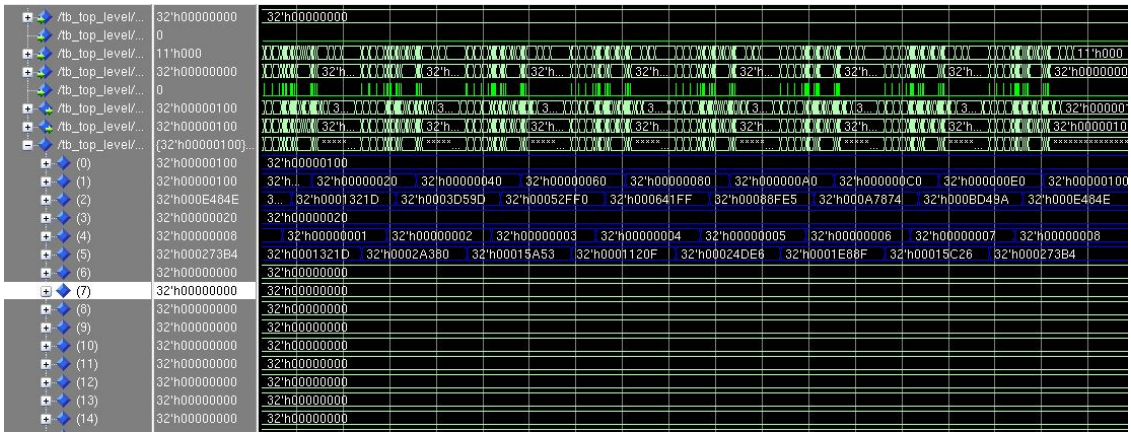


Figure 4.7: ModelSIM waveform example

4.2.2 ABFT

ABFT is different respect to the others three cods. It requires three different kernel calls. It can not be simulated as the others. The main problems are:

- The three kernel-calls require different parametrises.

- The input matrices of MatrixMul depends on the first two calls.
- Simulation wants just one .mif file.

In the following the main problems will be explained.

Issues and solutions

Each kernel was passed to Nsight and `"transformed"` in binary codes. Several PichBench files were produced in order to be suited to the needs of each call. Then the first two calls were simulated and the two output files were conserved.

To generate a single .mif file a python code was used. `ABunion.py` is a python code that takes as input the results of the first two calls, renamed as `rdataA_mem.txt` and `rdataB_mem.txt`. The output is a suitable memory for the simulator, so global_mem.mif file.

The new memory is different both the matrix-A's and matrix-B's memory. For this reason the pickBench file requires new addresses in the constant part, that point to the rearranged locations. Furthermore the first two calls transform the input matrices, adding a row and a column respectively. At the beginning the situation was:

$$dim(A) = dim(B) = dim(C) = 8 * 8 \quad \rightarrow \textit{64 threads}$$

But now:

$$dim(A) = 9 * 8, dim(B) = 8 * 9, dim(C) = 9 * 9 \quad \rightarrow 81 threads$$

ABFT is the only code that has a bigger C-matrix. Also in this case just one block was used, but the number of threads was increased.

As for the others codes, the simulation was performed and the result was checked.

4.3 Fault Injection

The simulation proves only that the code works in the right mode. To know how good the code is against fault injection fault is needed.

Faults can occurs in a random position of the vector registers files, but to have an effect it must occur in an used register.

Fist step was to collect all the register used by each code per each vector registers file of the FlexGrip. A list of the possible positions in which a fault could generate an error was written. It is importance to notice that faults are evaluated only at path level, the injection of fault in the control unit is not a topic of this thesis.

For example:

Consider a code with 4 threads per vector registers file, where each thread uses 6 registers. Each register is a 32-bits, so 32 position where a fault can occurs.

32*6*4*8= 6144 possible position

The result is multiply by 8 because FlexGrip is composed by 8 vector registers file.

In the presented example 6144 are the possible positions for a fault, but stuck-at can be 0 or 1. So at the end 6144*2=12288 faults to examine.

These positions in hardware are represented by signals, as the following image shown. 4.8

```
1  sa1 /uGPGPU/uStreamingMultiProcessor/gRegisterFile(1)\
2  /uRegisterFile/dp_regfile_inst/mem(0)(26) -
3  sa1 /uGPGPU/uStreamingMultiProcessor/gRegisterFile(1)\
4  /uRegisterFile/dp_regfile_inst/mem(0)(25) -
5  sa1 /uGPGPU/uStreamingMultiProcessor/gRegisterFile(1)\
6  /uRegisterFile/dp_regfile_inst/mem(0)(24) -
```

Figure 4.8: Signal definitions

4.3.1 Injection

The main idea is to inject a fault at the time in one of the possible signals of the list, simulate the affected code, and compare the resulted memory with a non corrupted memory.[**Ontheevaluationofsoft-errors**].

A non corrupted memory(`Golden Memory`) is the result of a `Golden Simulation`. A Golden Simulation is a simulation as previously described, where no fault occurs. 4.9

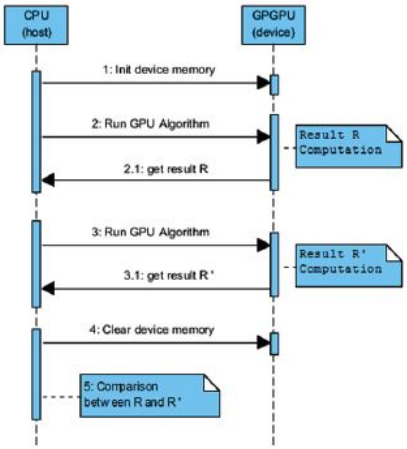


Figure 4.9: Injection process

To simulate a stuck-at 1 (or 0) one bit at simulation is forced to 1 (or 0 in the other case). The code is simulated again but this time one signal is corrupted. Comparing the Golden Memory and the resulted one possible errors can be detected.

This process is executed multiple time as the number of the signals in the list. A python code, named `Script_Fault_Simulator.py` deals with all these steps.

4.3.2 ABFT

ABFT used the same python script but with some changes. Changes are due to the presence of three kernel-calls.

Injection faults in the ABFT can be done in two different ways.

First method

As said in GPGPU's architecture, different streaming multiprocessors(SM) are available. If it is supposed that the three kernel-calls are executed by different SM, it is reasonable think that just one of the three is affected by a faults, for simplicity the faulted one is the last kernel-call, the matrixMul.

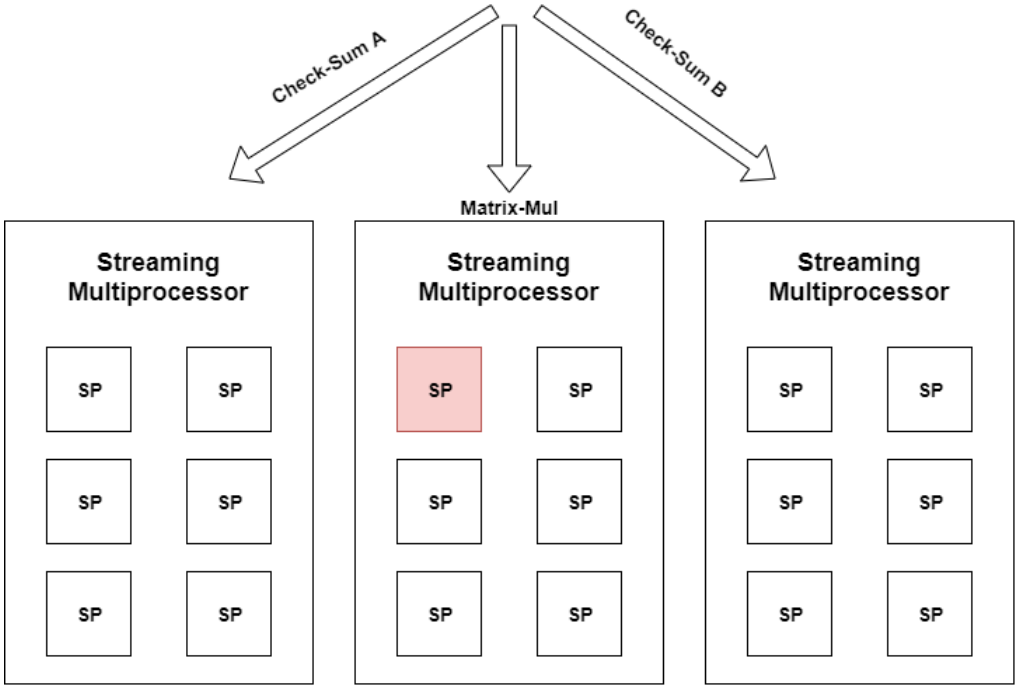


Figure 4.10: Distribution of kernels

In this case, the easiest one, the injection fault is performed only the matrixMull kernel. See 4.10.

In the first phase the golden memory is created, this stage is equal to the steps done in the simulation. The check-sum kernels are executed without any bits forced. The two memory results are united in a single memory for the input of the third

kernel. At this point the Golden memory for the ABFT is ready. It is remembered that ABFT means the union of the three kernels.

The second phase, as for the others codes, it is the injection fault. It is chosen a signal at simulation between the targets of the matrixMull kernel. The signal is forced to 0 or 1, and the fault simulation is performed. Golden and faulted memory are compared.

The process to generate a faulted memory in the ABFT can be summarized as follow:

1. Initialize simulator with MatrixMul's parameters.
2. Inject fault in MatrixMul.
3. Collect faulted memory MatrixMul(i.e the ABFT's result).

In the first point the input matrices are obtained as during the realization of the Golden Memory, just calling the ABUnion function using the golden results of Check-sum A and B.

Second method

In this case, instead, it is supposed that the three calls are executed by the same streaming multiprocessor. This means that if a fault is present, all the calls will present an error. It is importance notice that an error in the first two kernels is propagated in the third call. This is due to the fact that MatrixMul directly depends by the outputs of the first two calls.

The first phase to generate the Golden memory is equal to the first method. Instead generate a stuck-at in all the kernels is more complex. A signal that can affect MatriMul is chosen and stuck. Signals are chosen from the MatrixMul'list because this kernel uses more registers than the others two kernels, therefore a signal presented in it, it is also presented in the others two.

The process to generate a faulted memory in the ABFT can be summarized as follow:

1. Initialize simulator with Check-sum A's parameters.
2. Inject fault in Check-sum A.
3. Collect faulted memory A.
4. Initialize simulator with Check-sum B's parameters.
5. Inject fault in Check-sum B.
6. Collect faulted memory B.

7. Call function `UnionAB` to realize input memory's file for MatrixMul.
8. Initialize simulator with MatrixMul's parameters.
9. Inject fault in MatrixMul.
10. Collect faulted memory MatrixMul(i.e the ABFT's result).

From the list it can be seen how the number of steps is longer than the other case. In fact each time it is injected a fault in a different kernel, the parameters must be initialized again to compute the new simulation. Must be remembered that an injection fault is realized as a normal simulation with a bit forced to 0 or 1, so it is mandatory doing the steps described in the Simulation section.

As explained before, the result of the MatrixMul kernel depends by the first two. This implies not only that a faulty value in matrix A or B will be presented in C, but also that a crash in one or both the two inputs matrices will create a crash in the C matrix too.

See the figure 4.11.

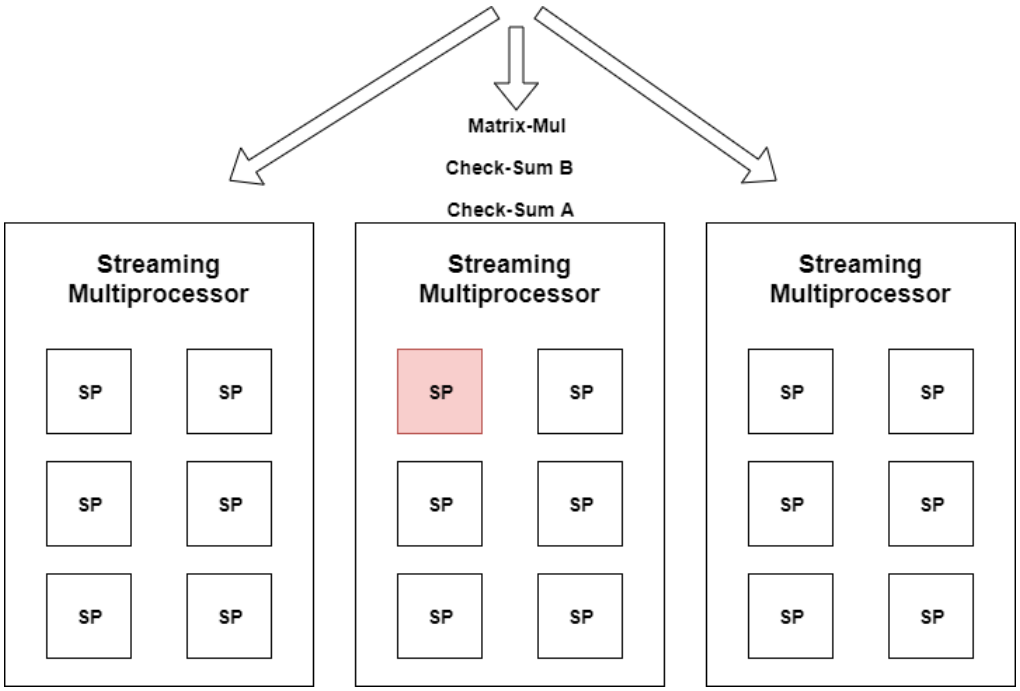


Figure 4.11: Presented case

4.4 Results

Errors produced during the injection fault simulation, can be divided in three main categories.

- Halt - This kind of error produce a crash in the code, therefore a crash in the simulation. It can be produced by forcing a bit in an address, so change the pointed destination. This produces a failed read or writing, making the system crashes.
- Time Out - Sometimes a fault generates an error that increases the simulation time, but at the end the system reaches the end with no errors in the memory.
- Silent Data Corruption (SDC) - When an error does not make crashes and reaches the end, can produce some errors in the resulted values. This kind of error is the target of this thesis.

In the following the experimental result of each codes is presented. It is a must to say that values of SDC is not so accurate, as it is subjected to data dependency. This means that if it is injected a stuck in a bit equal to the stucked value, this will not produce any error.

4.4.1 Matrix Multiplication

To verify the validity of the redundant codes, a matrix multiplication without any kind of protection was subjected to injection fault simulation. Obviously matrix multiplication is the easiest code, and it is part of each of the others codes.

Faults injected	**SDC**	**FC**	**Registers**	**Time (ns)**	**Instructions**
37376	19202	51.4	10	860270	48

In the above table the main relevant parameters in this thesis are reported.

- N°fault injected - It is the number of positions taken into account during the simulation.
- SDC - This are the SDC founded.
- FC - It is the partial fault coverage, it is computed as FC = SDC/ N°Fault injected * 100
- Registers - This is the number of registers used.
- Timing - It is the time used to compute the result.
- N°Instructions - It is the number of instructions in binary language used.

During each simulation it has been taken into account the time and the number of registers and instructions.

The number of registers implies the magnitude of the signals, therefore the number of possible location for a fault. Instead, instructions are taken into account to know the space occupied in the memory.

4.4.2 DWC V1

Faults injected	SDC	FC	Registers	Time (ns)	Instructions
53248	26448	49.7	13	1540390	59

Comparing the table of Matrix Multiplication and the above one, can be seen that the number of instructions and registers are increased. Due to the extra code added to protect the matrix multiplication(i.e the DWC implementation).

As the implementation of the Doubled with comparison is done to **high level**, it protects only the matrix's result without taking into account which registers are used. Therefore if a fault affects a register used as address, an halt error occurs. This problem is common in all the codes.

Focusing on the `SDC` errors, it can be seen that the improving of the Fault Coverage (FC) is not so better that the matrix multiplication. This is due to the fact that sometimes faults affect register used for multiple purpose. In the following will be presented the purpose of each register to better understand the concept.

- R0 - Load IndexThread, Load First limit of thread, Load **Input matrices B**
- R1 - Load Second limit of thread, Load **Input matrices A**
- R2 - Load Address matrix B1
- R3 - **Load limit of the For-statement**
- R4 - Load Address matrix C2
- R5 - Load Address matrix C1
- R6 - Load Address matrix B2
- R7 - Load Address matrix A1
- R8 - **Value used to increment the B's address**
- R9 - **Loop's counter**
- R10 - Load Address matrix A2
- R11 - **Temporary Store C1 and C2's results.**
- R12 - **Temporary values**

DWC computes two times the C-matrix (C1,C2), using the same inputs replicated two times(A1,A2,B1,B2). At the end C1 and C2 are compared, and in match case the C1 result is stored.

As it can be seen, some registers are more subjected to create errors than others. In bold are highlighted the main problems. In fact the real problem is that the

delicate registers are common in both the computing of C1 and C2. This means that C1 and C2 are both affected in the same way, therefore the comparison will produce a match. Comparing the faulted memory and the golden one, an SDC error is produced.

For example:

Register 0 is used to load from the global memory the input value of both A1 and A2. Injecting a fault in any bit of this registers will produce an error in the matrix multiplication for both C1 and C2. The same happens for register 0, 11 and 12.

Register 8 is used to increment the address of B-matrices (i.e take the next value). This register can create both halt and SDC errors. If the injection creates a valid address, a wrong value will be loaded for both C1 and C2, but the multiplication will finish. Instead if a non-valid address is produced an halt error is generated.

4.4.3 DWC V2

Faults injected	SDC	FC	Registers	Time (ns)	Instructions
40960	22412	54.7	10	1165320	54

As the table shows, the second version of DWC is the fastest between all the presented codes. Due to the fact that it uses less accesses to global memory than the others and works with local memory.

Unfortunately the coverage is worst than all the studied cases, even of the base case, the matrix multiplication code.

In the following how the compiler used registers is presented:

- R0 - ThreadIndex, Store address.
- R1 - Load input from B, some calculation.
- R2 - Load input from A.
- R3 - **Temporary result C1, Final C1 to store.**
- R4 - Loop's counter.
- R5 - Value used to increment the B's address.
- R6 - Load address of B.
- R7 - Load limit of the For-statement.
- R8 - Load address of A.
- R9 - Temporary operation.

In the previous chapter was explained that instead working with four matrices loaded from the global memory, only two loads are done. The two loads are used by two local temporary variables to compute the matrix-C's cells. In the list it can be seen that only one temporary variable is presented. This means that only one computation is performed. The issue is caused by the compiler. The compiler optimized the instructions, therefore it deleted the second computation, as it is equal to the other computation. The elimination of the second temporary variable means that no-comparison is performed. This explain why the coverage is so high.

This kind of code can not be used in high level (i.e writing only the CUDA code), but only at low level working with registers directly.

4.4.4 TMR

Faults injected	SDC	FC	Registers	Time (ns)	Instructions
61440	13870	22.6	19	1658370	73

TMR is the slowest and the longest, in term of execution, between the redundant code.

- R0 - ThreadIndex, Load input B1, **Voter**, **Store the final value of C**.
- R1 - Limit for ThreadIndex, Load input A1, **Voter**.
- R2 - Limit for ThreadIndex, Load input A2.
- R3 - Load input B2
- R4 - Load input A3
- R5 - Load input B3
- R6 - Load address matrix A1
- R7 - Load address matrix A2
- R8 - Load address matrix A3
- R9 - Load address matrix B3
- R10 - Load address matrix B2
- R11 - Load address matrix B1
- R12 - **Value used to increment the B's address**
- R13 - **Loop's counter.**
- R14 - **Load limit of the For-statement.**

- R15 - **Voter**
- R16 - **Voter**
- R17 - **Voter**
- R18 - Address to store final C-matrix.

As for the DWC version one, the delicate registers are in bold. In the TMR values from the global memory are loaded in six different local registers, therefore the issue of the DWC's version one is not presented. Instead, registers 12, 13 and 14 have the same issues presented for the first case.

The main difference is the presence of the voter. The three results are computed in three different local variables and the comparison is done two by two. The registers used as voter are common for all the three cases. Therefore a fault in one of these produces an SDC error.

Moreover the register 0 is used for voter and store for the final value. This means that R0 is the most delicate register.

4.4.5 ABFT

The ABFT is the most complex and with more delicate registers than the other cases, due to the larger amount of instructions than for the other techniques, as follows:

Faults injected	**SDC**	**FC**	**Registers**	**Time (ns)**	**Instructions**
92160	23953	26	22	4085470	247

As already said in the section "Injection Fault", signals used are taken by the matrix multiplication kernel. But the fault are injected also in the first two kernels, therefore some cases could create a crash before fault simulation in the matrix-multiplication starts.

How registers are used by Check-sum-A and B are presented.

- R0 - **ThreadIndex**, **Load destination address.**
- R1 - **Load input address.**
- R2 - Temporary values
- R3 - Value used to increment the A's address
- R4 - Loop's counter.
- R5 - Load input value.

In the following Check-sum-B is presented.

- R0 - **ThreadIndex**, **Load input address.**
- R1 - Loop's counter.
- R2 - Limit
- R3 - Temporary values.
- R4 - Load input value.

Registers that can cause a crash before the matrix-multiplication is performed, are underline in bold. Obviously an errors in both the two matrices will be propagated in the third process.

The third kernel is really larger, and registers are used for different purposes. In the previous section were explained that this kernel can be divided in three/four phases, registers changes their purpose according to the phases.

Purpose of the registers are presented divided in the three phase.

First phase - Matrix multiplication

- R0 - ThreadIndex, **Limit**, **Load input matrix B**
- R1 - **Limit** , **Load input matrix A**
- R2 - Load input address A
- R4 - **Loop's counter**
- R5 - **Value used to increment the A's address**
- R6 - **For's limit**
- R7 - Load input address B
- R9 - Load input address to store
- R10 - **Temporary values**, **Final value to store**

This phase is the normal matrix multiplication, therefore any "protection" is present. If the first two kernels do not present any errors, the ones that can occur in this phase (considering only SDC errors) can be founded in the next phases and they can be corrected. Obviously an halt error causes the crash, therefore can not be fixed.

Second phase A - Comparison vector one creation

- R0 - **Load input address**
- R1 - Limit for thread that can access to this part of code, **Set error**
- R5 - **Temporary values**
- R6 - **Value used to increment the address**
- R7 - **Load store address**
- R8 - **Loop's counter**
- R9 - **Load input value**

In this phase one of the two comparison vectors are generated. An error in this phase is not too dangerous, as it will be fixed during the comparison phase. In fact if an a mismatch is found between the last row/column and the two vectors, a restore is performed to correct the matrix. Therefore if the issue affect only this process and not a value in the matrix, the restore is performed with not a real necessity. The **only exception** is done for register 1, as it is used to notified an error.

Similarly for the next registers.

Second phase B - Comparison vector two creation

What said for the previous registers is still valid for these ones. In this case the exception is represented by the register 0.

- R0 - Limit for thread that can access to this part of code, **Set error**
- R1 - **Loop's counter**
- R2 - **Load input address**
- R6 - **Load input to store values**
- R7 - **Load input value**
- R8 - **Loop's counter**

Third phase - Comparison

- R0 - Thread used
- R3 - **Temporary values**
- R4 - **Temporary values**

In this phase the comparison is performed, therefore it is the most critical phase. As a fault in the "voter" creates a not-detected error.

The comparison phase uses more registers than presented ones. As the study of the registers were performed in the simulation phase, when no errors occur.

Some registers are more delicate than others, but the most critical is register 0. This register is involved in all the process in more purpose. Moreover in the last phase only thread (0,0) works, but if a fault affect register zero, that represents the thread index, nobody performs the comparison.

In conclusion this code requires a lot of time and space in memory but it reaches a good result.

Chapter 5

Conclusion

Looking the graph it can be seen that the TMR reaches the best result. It reaches the 22 percentage spending less execution time than the other technique using a moderate space in memory. Obviously the DWC required less time than TMR but the coverage is only of the 50 percentage. The second version of the DWC is failed, and therefore it will not be commented. The ABFT reaches a good coverage, even if worst than TMR. This can be due to the fact that the others techniques start with two corrected matrices and only one kernel was affected by faults. The ABFT can have issues also in the beginning matrices, therefore the matrix multiplication starts with wrong values, so obviously also the result is wrong. Despite the problems of ABFT, this has achieved excellent coverage, slightly lower than the TMR. Probably considering the first implementation of the ABFT the coverage could have been better.

Some improvements and future tests can be made. For example it could rewrite the DWC2 code in PTX, avoiding the compiler optimizations, or in the ABFT try to protect register 0 which is the most affected by errors. Moreover also transient faults can be taken into exam repeating the tests.

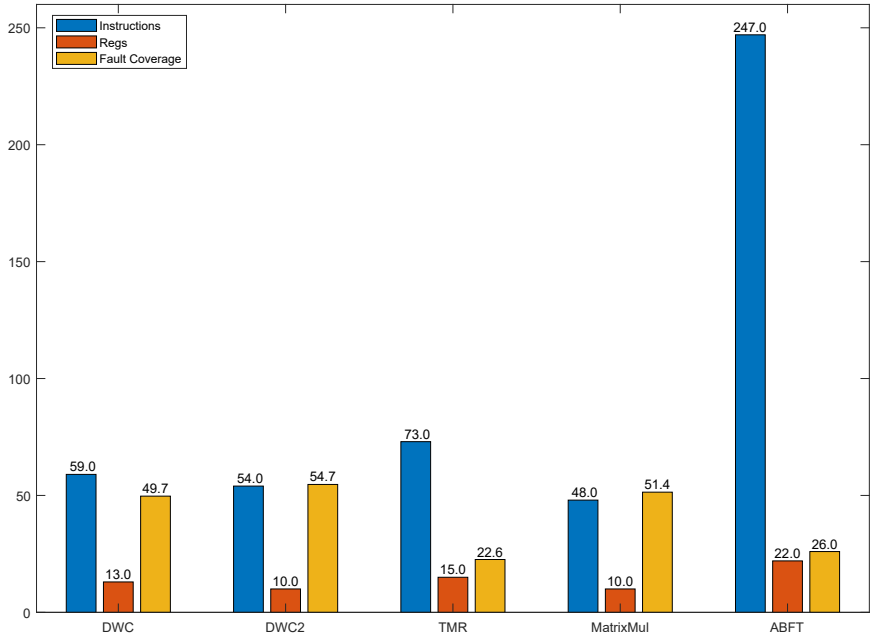


Figure 5.1: Overview 1

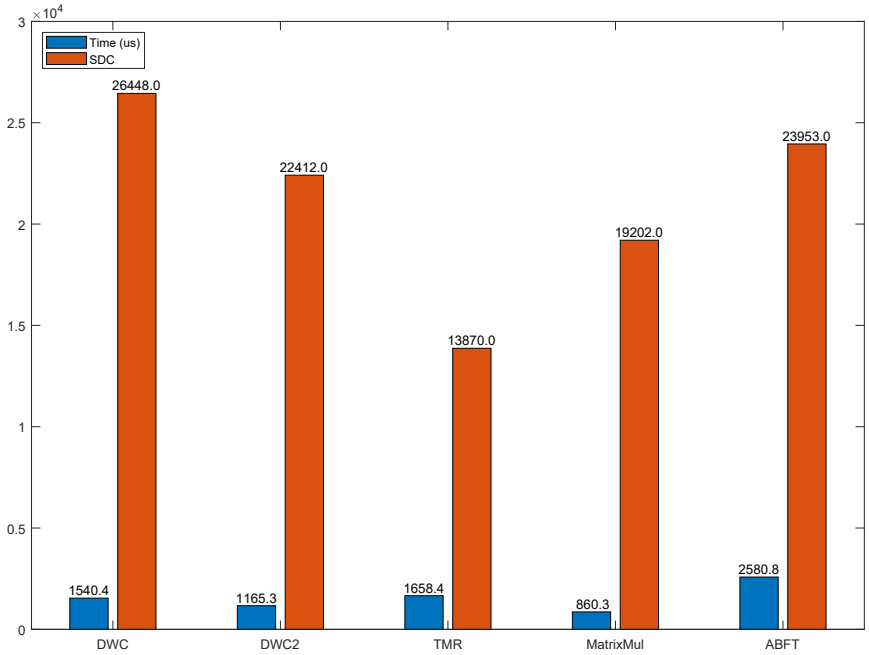


Figure 5.2: Overview 2

Bibliography

[1] URL: http://stevendebock.blogspot.com/2013/08/the-graphics-pipeline.html.

[2] URL: %https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=2ahUKEwjFwouG6ofmAhVM1qYKHThAAwkQFjAAegQIBBAC&url=http%3A%2F%2Fwww.cs.cornell.edu%2Fcourses%2Fcs4620%2F2013fa%2Flectures%2F09rasterization.pdf&usg=AOvVaw0yImHyg0KA21F1AhmyW1g8.

[3] B. Baykant ALAGOZ. "Hierarchical Triple-Modular Redundancy (H-TMR) Network For Digital Systems". In: *ding* (2007).

[4] Kevin Andryc, Murtaza Merchant, and Russell Tessier. "FlexGrip: A soft GPGPU for FPGAs". In: *2013 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2013. DOI: 10.1109/fpt.2013.6718358.

[5] Claus Braun. "Algorithm-Based Fault Tolerance for Matrix Operations on Graphics Processing Units: Analysis and Extension to Autonomous Operation". PhD thesis. Institut für Technische Informatik der Universität Stuttgart, 2015.

[6] Antonio Carzaniga et al. "Automatic Workarounds". In: *ACM Exploiting the Intrinsic Redundancy of Web Applications* 24.3 (2015), pp. 1–42. DOI: 10.1145/2755970.

[7] Shane Cook. *CUDA programming - A Developer's guide to parallel computing with GPUs*. 2008.

[8] Martin Dimitrov, Mike Mantor, and Huiyang Zhou. "Understanding software approaches for GPGPU reliability". In: *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-2*. ACM Press, 2009. DOI: 10.1145/1513895.1513907.

[9] Chong Ding et al. "Matrix Multiplication on GPUs with On-Line Fault Tolerance". In: *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*. IEEE, 2011. DOI: 10.1109/ispa.2011.50.

[10] R. E. Lyons W. Vanderkul k. "The Use of Triple-Modular Redundancy to Improve Computer Reliability". In: *IBM JOURNAL* (1962).

[11] Romano Laprie Kanoon. "Fault-tolerant design techniques". In: *Fault-tolerant design techniques*. 2008.

[12] Erik Lindholm et al. "NVIDIA Tesla: A Unified Graphics and Computing Architecture". In: *IEEE Micro* 28.2 (2008), pp. 39–55. DOI: 10.1109/mm.2008.31.

[13] Andrea Mattavelli. "Software Redundancy: What, Where, How". PhD thesis. Politecnico di Torino, 2016.

[14] P. Rech et al. "An Efficient and Experimentally Tuned Software-Based Hardening Strategy for Matrix Multiplication on GPUs". In: *IEEE Transactions on Nuclear Science* 60.4 (2013), pp. 2797–2804. DOI: 10.1109/tns.2013.2252625.

[15] Gianluca Roascio. "Analysis and extension of an open-source VHDL model of a General-Purpose GPU". PhD thesis. POLITECNICO DI TORINO DEPARTMENT OF CONTROL and COMPUTER ENGINEERING (DAUIN), 2018.

[16] Nicholas Wilt. *The CUDA handbook - Acomprehensive Guide to GPU Programming*. 2008.