Collegio di Ingegneria Informatica, del Cinema e Meccatronica

Class LM-25 : Mechatronic Engineering

A simulator implementing NVIDIA kepler GPU for ADAS applications

Mohammed Faisal Helmy Elrashedy

Supervisors

Prof S. Di Carlo

Prof A. Vallero

23rd October 2019

Summary

In this thesis, the reliability of the General Purpose Graphics Processing Unit (GPGPU) NVIDIA Kepler architecture is evaluated. We will use the Multi2sim Simulation framework to assess the reliability of the Kepler architecture; moreover, a fault injector is developed along with ACE analysis to evaluate the performance and the impact of such faults to supercomputing in mission-critical applications.

Acknowledgements

I would like to acknowledge prof. Allesandro Vallero for his continuous support and dedication; I wish him all the best in his path. Moreover, I would like to thank Ms. Silvia Kuehl for her brilliant assistance; I wish her continuous development and a satisfying career.

Contents

List of Figures

Chapter 1

Background

In this chapter, we will discuss the main principles of **GPUs**, like parallel computing. Moreover, we will discuss the building blocks of our analysis, which are: **NVIDIA Kepler**, **multi2sim**, and reliability for **micro-architecture**.

1.1 General Purpose Graphics Processing

The General Purpose Graphics Processing Unit (**GPGPU**) alters the path of general purpose computing by performing parallel computing faster than CPUs, as shown in 1.1

GPGPUs enables the performance of **non-graphic** calculations and operations, as fast as real-time MPEG videos. For instance, **GPUs** can be used to compute fast-Fourier transform functions, scientific computing as Monte Carlo simulation and weather forecasting, and neural networks [1]

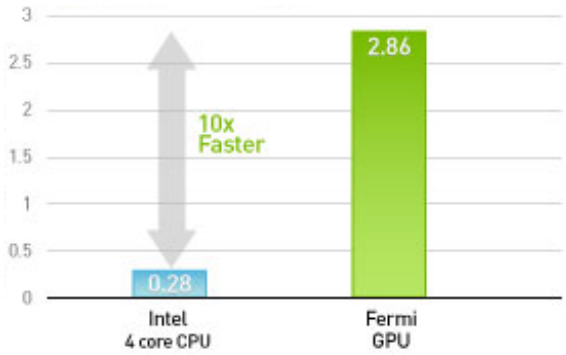

Figure 1.1: Numbers of 512x512 images processed by Deep Neuron Network per second

1.1.1 History

In 2001, **NVIDIA** and **ATI** revolutionized graphics computing by allowing their **GPUs** to be programmed. Developers could write limited programs to execute at real-time level, like bump mapping or shadowing.

Microsoft played an essential role in programming **GPUs**, by introducing DirectX 8.0, which provides 3D graphics APIs for Windows and Xbox. DirectX offered fixed-function pipelines. Therefore programmers had limitations programming **GPGPUs**. DirectX 9 secured the next big step in the progress of graphics computing, by enlarging instruction sets for pixel shading, and by increasing the mathematical precision up to a 128-bits floating-point from a 8-bits precision. C for Graphics **Cg**, a creation of both **NVIDIA** and Microsoft, is the programming language in which graphics units re programmed. Figure 1.2 and 1.3 illustrates the architecture before and after **GPUs**.[1]

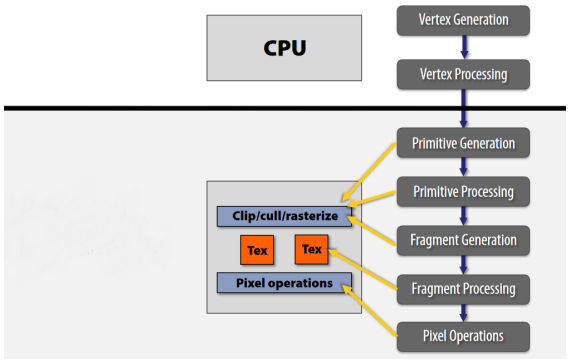

Figure 1.2: PC-3D graphics pre-1999

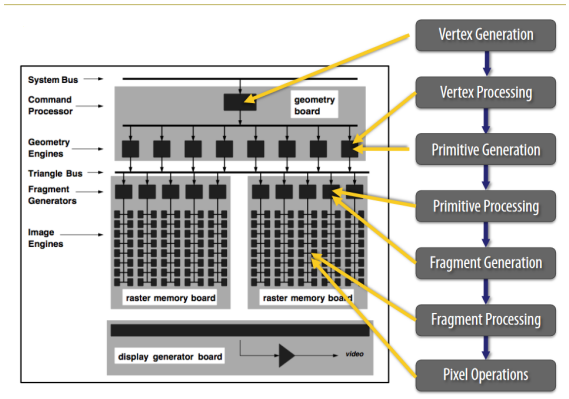

Figure 1.3: Silicon Graphics RealityEngine 1993

1.1.2 GPGPUs Architecture

Today's **GPUs** are flexible: they do support high-level programming and they have a precision of 32-bit floating-point. Ultimately, **GPGPUs** are available to developers as, to some extent, **co-processors**.

To program a NVIDIA **GPGPUs**, it is necessary to write a piece of code called **Kernel**, which is the running function on each streaming pipeline of the **GPGPUs**, as in figure 1.4 shows a program to copy memory from device to host and the vise versa .In graphics domain, a Kernel would be defined as a 'shader'. Using APIs like **OpenGL, OpenCL, and CUDA**, **Kernel** and **shader** functions can be accessed from the host: **GPU's** main memory or in-chip memory.

```
  MY_API void kernelIncrement(int * data)
{
     int workItemId = threadIdx.x+blockIdx.x*blockDim.x;
     data[workItemId]++;
}
cudaMemcpy(gpuData, hostData, n, cudaMemcpyHostToDevice);
kernelIncrement<<<128,128>>>(gpuData);
cudaMemcpy(hostData, gpuData, n, cudaMemcpyDeviceToHost);
```

Figure 1.4: Kernel Code example

The central concept of **GPGPUs** is the Single Instruction Multiple Data (**SIMD**), which allows parallelism. **SMIDs** creates the hardware, while the API introduces them as multi-core CPU to ease writing programs. Figure 1.5 shows a simple parallel execution of a program where there is a single instruction the addition and vectorized data a and b and results in vectorized data c.

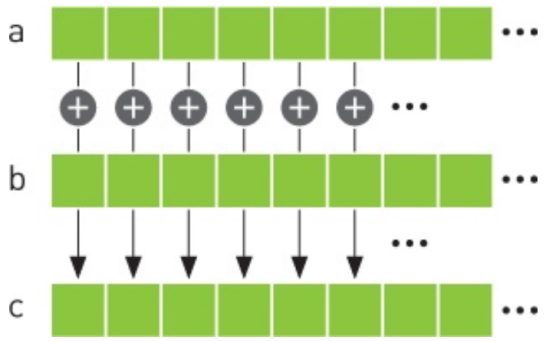

Figure 1.5: A simple parallel addition program

In writing vectorized codes, for example, APIs like **CUDA** have online-compilation

that get your **non-SIMD**, scalar-simple, and map it into **SIMD** units. These APIs support multiple architectures: once the code is written, it can run on a **NVIDIA** architecture.

1.1.3 NVIDIA Architecture

Streaming multiprocessor**SMs** are the building blocks of **GPUs** and execute the actual computation. As illustrated in figure 1.6, each **SM** possesses control units, registers, execution pipelines, and caches.

Figure 1.6: The main architecture of NVIDIA

Each **SM** has multiple **CUDA** cores which are displayed in figure1.9 . **CUDA** core holds the floating-point unit, Fused multiply-add, logic unit, Move, compare unit, and, finally, the branch unit. **SM** holds individual function units such as cos, sin, and tan. In addition, it carries shared memories, L1 & L2 cache, and thousands of 32-bit registers, where the **SM** read from, as illustrated in figures 1.7 and 1.8 respectively. Despite their limited size, **SM** memories are extremely fast. Programmers can select which cache to use, whether L1 or shared memory.
L1 cache is designed for hardware use as register spilling, and should not be used as CPU caches. By contrast, shared memory acts as a scratchpad, with data accesses that must be synchronized. Finally, unified L2 cache is fast, coherent, and shares data across all the **GPU** cores.

Figure 1.7: Inside Streaming Multi Processor SM

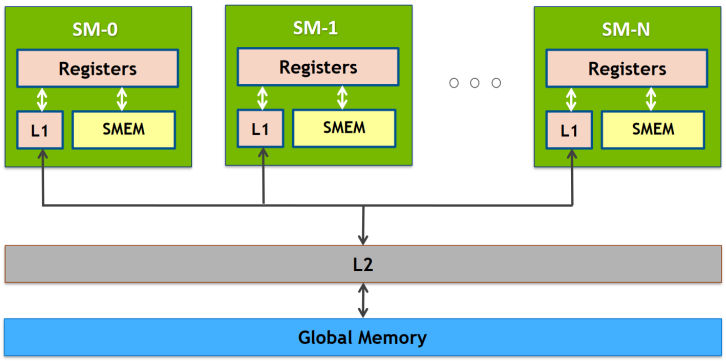

Figure 1.8: NVIDIA Memory Hierarchy

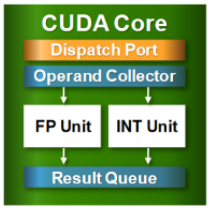

Figure 1.9: Cuda Core inside

1.1.4 NVIDIA Execution Model

Threads are the main component of the software running on **GPU**, handled by the Scalar processor **(CUDA)**. Whereas the whole **Thread block** executes on multiprocessors, each **Thread block** consists of **warps** that execute in parallel. This execution called **Single Instruction Multiple Thread (SIMT)**.

Thread blocks can stack on one **SM**, depending on the architecture or the available resources.

When the **Kernel** is launched, it executes as a **grid** of **Thread blocks**. Next, it runs on the **GPU**. Figure 1.10 shows how the SW runs on the **GPU**.

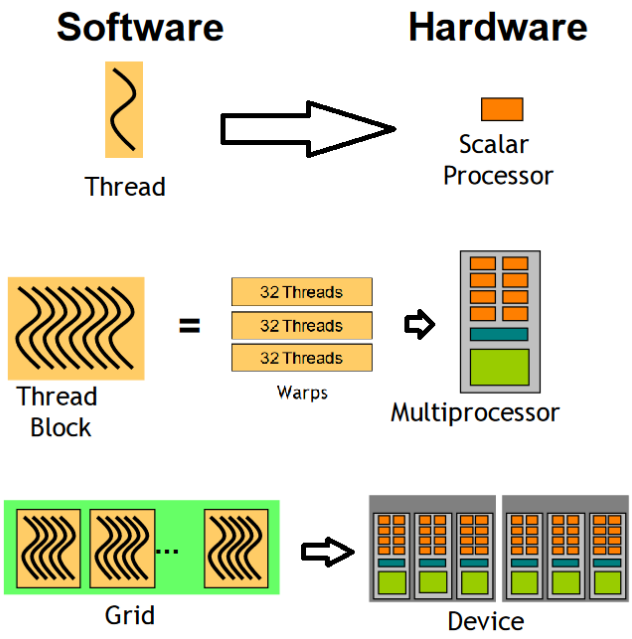

Figure 1.10: Software execution on GPU

1.2 Multi2Sim Simulation Framework

We are interested in micro-architectural simulators as multi2sim **(m2s)** because they allow us to evaluate the performance as power consumption and memory footprint in the early design stages. (m2s) is a simulator for CPU, **GPU**, and Heterogeneous systems, written in C and C++11. The Multi2Sim architecture consists of four independent software modules that interact with predefined interfaces, as illustrated in figure 1.11 The modules are:

- Module 1: Disassembler
- Module 2: Functional simulator

- Module 3: Detailed simulator
- Module 4: Visual Tool

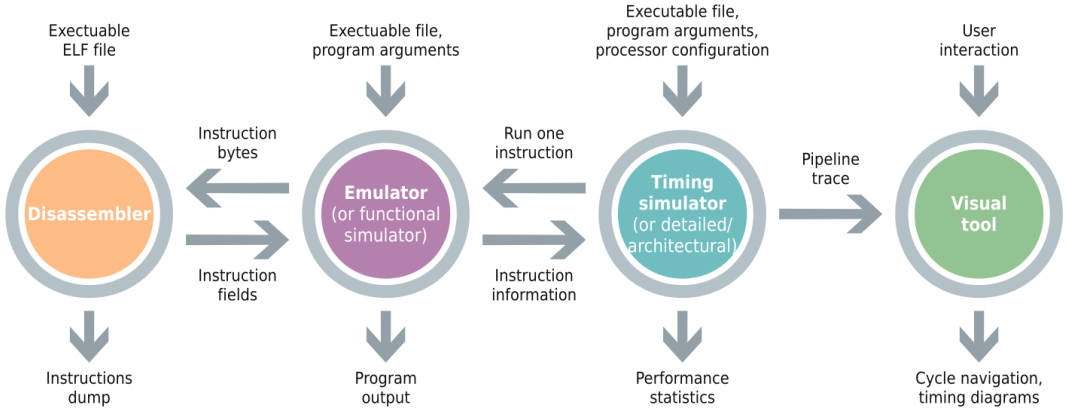

Figure 1.11: Multi2Sim's simulation model example

1.2.1 Architecture

Considering that every stage has its own specific function, it needs to propagate the data in the right manner for the following stage. Thus, a stage can not function properly without the preceding stages.

First, the **disassembler**'s purpose is to decode the instructions extracted from the guest program running into useful data. It can then be used for simulation purposes, such as operation codes, input/output operands, or immediate constants. As mentioned before, every module can work independently or as part of the whole simulation. In the first case, the disassembler would be interested only in the **instruction set architecture** (ISA) found in the binary file, where it dumps a text file including all the found instructions. While In the second case, the disassembler uses a buffer to iterate through each instruction, then maps every one accordingly to its field.

The second module consists of the **emulator** or **functional simulator**. Here, the simulator illudes the guest program that it is executing natively on a given micro-architecture. This occurs with the emulator tracking the state of the guest program and continuously updating the instructions, one by one, until the guest program ends. To do so, the emulator passes through four main steps:

- Reading the instruction;
- Decoding the instruction;

- Emulating the instruction and updating memory and registers;
- Incrementing instruction pointer (IP).

The most important aspect of the functional simulator is that it simulates timing cycles accurately. If the functional simulator is used alone, the program counter starts reading from the address of the first instruction — next, the simulation loops through the four steps mentioned above until the program terminates. If the simulator is used for the next stages, it provides all the data related to the emulated instructions. Figure 1.12 depicts the emulator's functioning for the 4-stage processor pipeline.

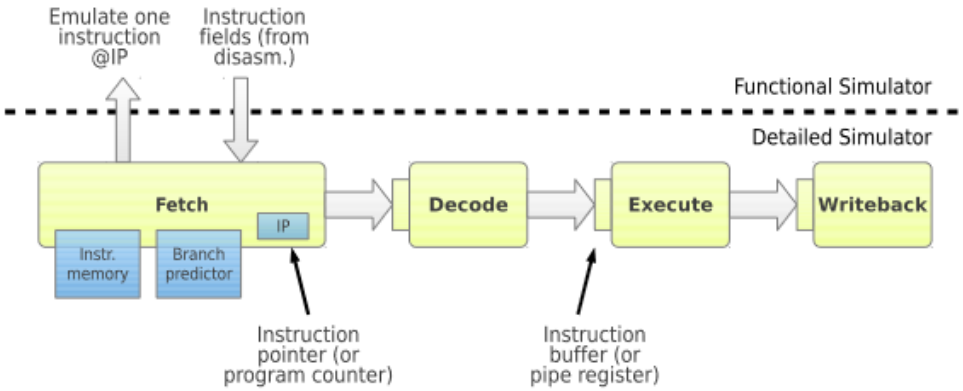

Figure 1.12: 4 stage processor pipeline,and the communication between the detailed and the functional simulators.

The third module involves a detailed simulator or timing simulation. In this stage, the hardware structures are modeled into pipeline stages, pipe registers, instruction queues, functional units, cache, and other needed structures. The simulator also tracks time accessibility for modeled structures. When the timing simulation detects free ports in the instruction memory, and free space in the fetch&decode pipe register - after the functional simulator sends all the required data from the current instruction - it requests the functional simulator to fetch new instruction. In summary, the timing simulator works to simulate one iteration of the program loop modeling one real-clock cycle. However, the simulator takes more time to simulate one cycle; It is the most accurate cycle compared to the execution on the Hardware.[6]

Finally, the visual tool provides visualization for each simulation cycle; that, however, will not be covered in this dissertation.

1.2.2 Operating Modes

There are two modes of operating m2s: Full-system and Application-only Emulation.
Full-System Emulation executes the full stack of software as if it was running on real hardware. It starts running the guest operating system (OS) form a disk image. Physical memory image with the values of the register file represents the state of the OS. The full-system emulator runs the guest OS and tracks I/O as if it was a virtual machine; its downside is the speed of running ISA. In fact, the Full system emulation is slow to feed other software components concerning timing simulation.

Application-only Emulation runs by removing the OS from the guest software stack. In this way, services are abstracted by the emulator, providing the application with initial memory image (program loading), and run-time communications between OS and the application. Program loading is simplified in the following steps: 1) Analyzing application ELF binary, especially ISA parts and initializing static data; 2) Initializing memory for the guest program by mapping ELF sections accordingly in virtual base addresses; 3) Initializing program stack by copying image program arguments and variables to the memory; 4) Assigning value of the architected file to the stack. After these steps, the emulation can start fetching the first ISA. When the guest's program requests an interrupt from the OS, the emulator collects all the data required by the interrupt and then update its internal state illuding the guest program that it runs natively.[6]

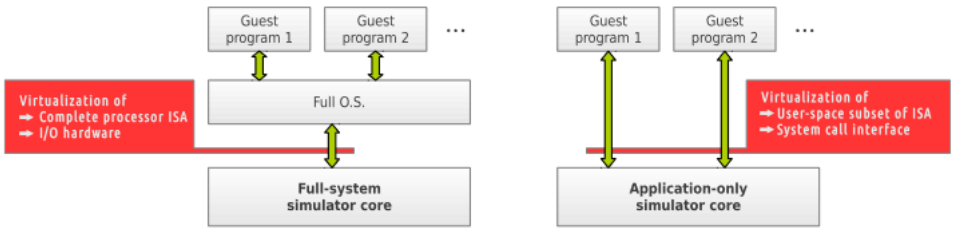

Figure 1.13: Full-system VS Application-only Emulation.

1.3 Vulnerability of GPGPUs

High-Energy particles bombarding **GPUs** is a concern for safety-critical systems such as automotive, aerospace, and medical devices.
When high energy neutrons or alpha particles hit sensitive nodes on the micro-architecture, the current produced can drive circuit simulators and cause transient

hardware errors. Transient errors are able to generate errors at the application level. In this thesis transient faults are modeled as single bit-flips in memory arrays. There are two different types of application and execution error: the first type is silent data corruption **(SDC)**, in which data propagates without the user noticing; the second type is detected unrecoverable error **(DUE)**, which makes the program crash while avoiding any corruption of data.[5]

Architectural vulnerability factor **(AVF)** **AVF** is the probability that a fault in a processor structure will result in a visible error in the final output of the program. **AVF** is defined as in equation 1.1. However, **AVF** nonetheless it is not an absolute factor: in fact, it varies depending on workload, time, and circuit dimensions. Being a time-dependent factor, it creates an user-visible error. [5]

$$AVF: \qquad = \frac{number\ of\ vulnerable\ bits\ instructure}{total\ number\ of\ bits\ in\ structure} \tag{1.1}$$

Chapter 2

Instrumentation

This chapter presents the instruments used and the setup implemented in the project. **NVIDIA Kepler** micro-architecture is evaluated for reliability using a Fault injector **(FI)** and **ACE analysis** which are built on the top of the **Multi2sim** framework.

2.1 NVIDIA Kepler

GPGPUs are constantly under development and object for improvement. This thesis adopts the **Kepler** architecture, acknowledging that several successors, such as Turing and Maxwell exist. However, our Analysis is easily extendable for the newer models mentioned.

Kepler was released in 2012 in three versions: **GK104, GK110,** and**GK210**. The last two models are explored in details in this thesis. **GK110** was originally designed for **Tesla.** Significant later features developed increased computing horse-power and decreased power footprint and consumption. GK110 and GK210 have up to **15 SM** units and six 64-bit memory controllers.

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110	KEPLER GK210
Compute Capability	2.0	2.1	3.0	3.5	3.7
Threads / Warp	32				
Max Threads / Thread Block	1024				
Max Warps / Multiprocessor	48		64		
Max Threads / Multiprocessor	1536		2048		
Max Thread Blocks / Multiprocessor	8		16		
32-bit Registers / Multiprocessor	32768		65536		131072
Max Registers / Thread Block	32768		65536		65536
Max Registers / Thread	63			255	
Max Shared Memory / Multiprocessor	48K				112K
Max Shared Memory / Thread Block	48K				
Max X Grid Dimension	2^16-1		2^32-1		
Hyper-Q	No			Yes	
Dynamic Parallelism	No			Yes	

Figure 2.1: KEPLER Improvements

2.1.1 Architecture

The **SMx** holds broad architectural improvements, enabling it for the double-precision workload. The new **SM** holds **192** single-precision equipped with single and double-precision arithmetic units, in addition to fused multiply-add **(FMA)**. **SMx** utilizes the principal **GPU clock** , which permits the increase of throughput outside of having multiple copies of execution units.

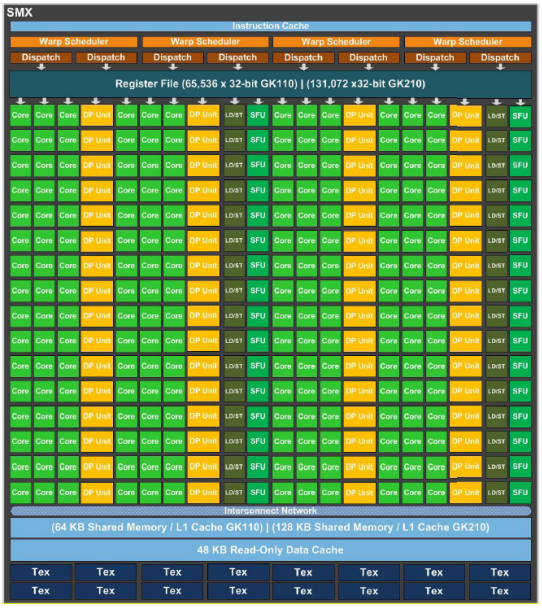

Figure 2.2: Kepler's SMx architecture

Quad Warp scheduler is the newly adapted technology in **Kepler SMx**. As mentioned in the first chapter, a **warp** forms for every 32 **Threads**. In **Kepler**, there are four **warps** and eight instruction dispatch units. Each cycle **warp** schedulers vouchsafe dispatching double instructions, permitting parallelism of double-precision instructions with others.[4]

Kepler performance significantly increases, permitting each **Thread** to access **255 Registers**, which in turn decreases memory spill to local memory. Within the same **warp**, **Threads** can exchange data using the new shuffle instructions. These can reduce the shared memory size needed by a **Thread**.

2.1.2 Memory Hierarchy

Kepler's **shared memory** and **L1** cache are *configurable*. IN GK210 there are 128 KB of adaptable memory, which can be used as **l1 cache** or **shared memory**.

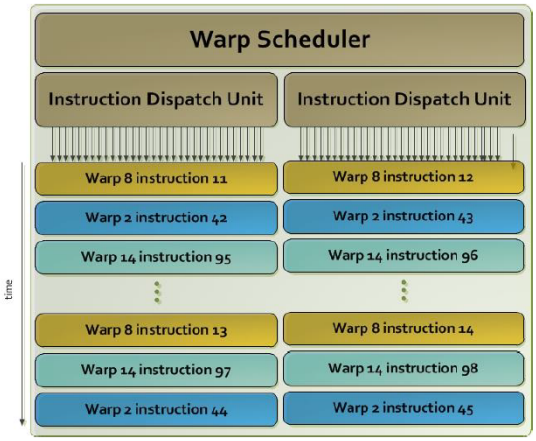

Figure 2.3: Kepler's Quad Warp Scheduler

Read-only data cache and **L2** cache have seen diverse improvements. A read-only **48kB** cache can be used to map data as textures, and it is optimized with compiler through *const__restirict* keyword. L2 cache was doubled in size in addition to bandwidth per clock cycle.

Kepler's memory Hierarchy is protected by a Single-Error Correct Double-Error Detect **(SECDED) ECC** code, while single-error correction protects the Read-only Data Cache through a parity bit.[4]

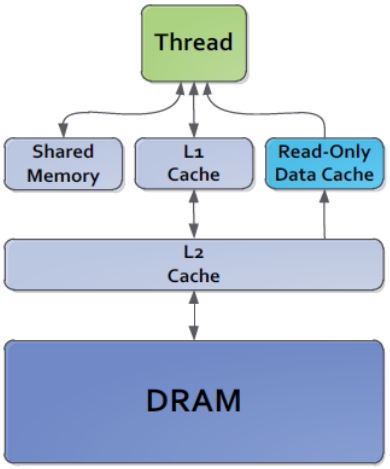

Figure 2.4: Kepler's Memory Hierarchy

2.2 KEPLER on MULTI2SIM

CUDAs applications running on **Multi2sim** are simplified into four entities, illustrated in figure 2.5.

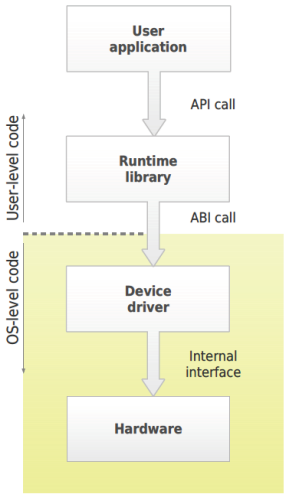

Figure 2.5: Kepler's SW modules

2.2.1 M2S Execution

Events Order On M2S:

- *cudaLaunchKernel* API is launched by the host program(x86 executable).
- Program calls are intercepted by the runtime library;
- Driver starts the **GPU** emulator, which interprets the API calls and reads the **Grid** values;
- The **GPU** emulator starts the simulation by initialing the loop.

2.2.2 Simulation Processes

In the simulation process for **KEPLER,** only the first **three** stages are available as shown in figure 2.6. There is no Visual tool.

Disassembler

The disassembler works on the same principles as those presented in chapter 1.2. Figure 2.7 illustrates an example of code translated in assembly.

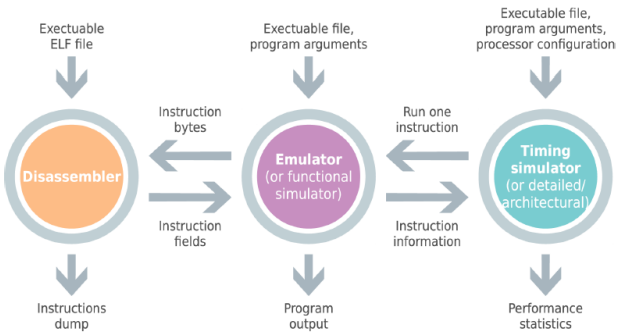

Figure 2.6: Multi2sim three stages for KEPLER

```
__global__ void vectorAdd(
        const float *A,
        const float *B,
        float *c,
        int numElements )
{
        int i = blockDim.x * blockIdx.x
                + threadIdx.x;
        if (i < numElements)
        {
             C[i] = A[i] + B[i];
        }
}
```

```
/*0008*/                MOV R1, c[0x0][0x44];
/*0010*/                S2R R0, SR_CTAID.X;
/*0018*/                S2R R3, SR_TID.X;
/*0020*/                IMAD R0, R0, c[0x0][0x28], R3;
/*0028*/                ISETP.GE.AND P0, PT, R0, c[0x0][0x14c], PT;
/*0030*/         @P0 BRA.U 0x78;
/*0038*/        @!P0 ISCADD R3, R0, c[0x0][0x140], 0x2;

/*0048*/        @!P0 ISCADD R2, R0, c[0x0][0x144], 0x2;
/*0050*/        @!P0 LD R3, [R3];
/*0058*/        @!P0 ISCADD R0, R0, c[0x0][0x148], 0x2;
/*0060*/        @!P0 LD R2, [R2];
/*0068*/        @!P0 FADD R3, R3, R2;
/*0070*/        @!P0 ST [R0], R3;
/*0078*/             MOV RZ, RZ;

/*0088*/             EXIT;
/*0090*/             BRA 0x90;
/*0098*/             NOP;
/*00a0*/             NOP;
/*00a8*/             NOP;
/*00b0*/             NOP;
/*00b8*/             NOP;
```

Figure 2.7: code that has been disassembled

Emulator

Emulation loop has two types of execution: the first one is **Thread block** execution, where **Thread blocks** execute randomly, with a policy of one **Thread block** at a time. This type of implementation is not relevant to the emulator. The second one is **Warp** execution, in which **warps** inside a **Thread block** execute in random order. The protocol to be respected is synchronization and one **warp** at a time [3].

Timing Simulation

After the **Grid** is created, **Thread blocks** are mapped by the scheduler into available **SMs**. Each **SM** accommodates four **warp pools** containing the **warps** of the assigned **thread blocks**. The **warps** to be executed are scheduled by the

front-end. The process develops under the following rationale:

- Instructions are fetched by the front-end, to be then decoded and sent to the execution units (that have 32 parallel units named lanes);
- The special functional unit, load-store unit, and integer math unit have one instance;
- Single precision units, double precision units, and branch units have multiple instances.

Figure2.9 shows the existing piping.

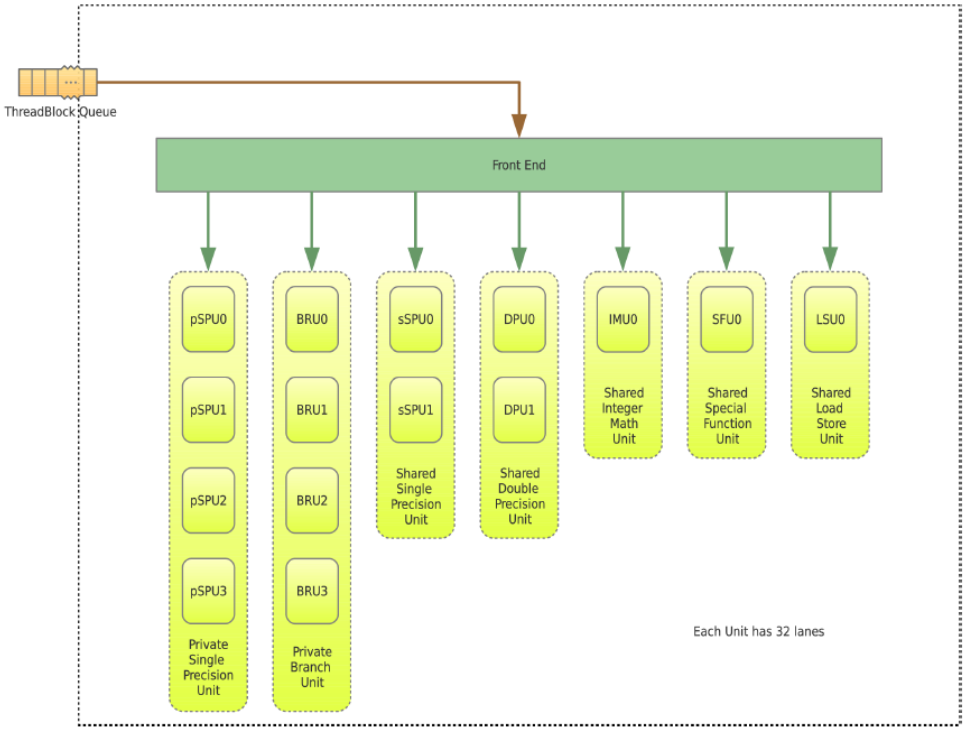

Figure 2.8: Execution unit for timing simulation architecture

2.2.3 Architecture and Piping

Front-end

The piping of the front-end works in the following sequence:

- There are four **warp pools** , each of them holding an assigned **Thread block**;

- All the four **warp pools** present their request to the instruction memory during the fetch stage, for each cycle;
- Dispatch stage is responsible for consuming the instructions from the buffers and sending them to the proper execution unit.

The interior of the front-end is shown in figure 2.9 below.

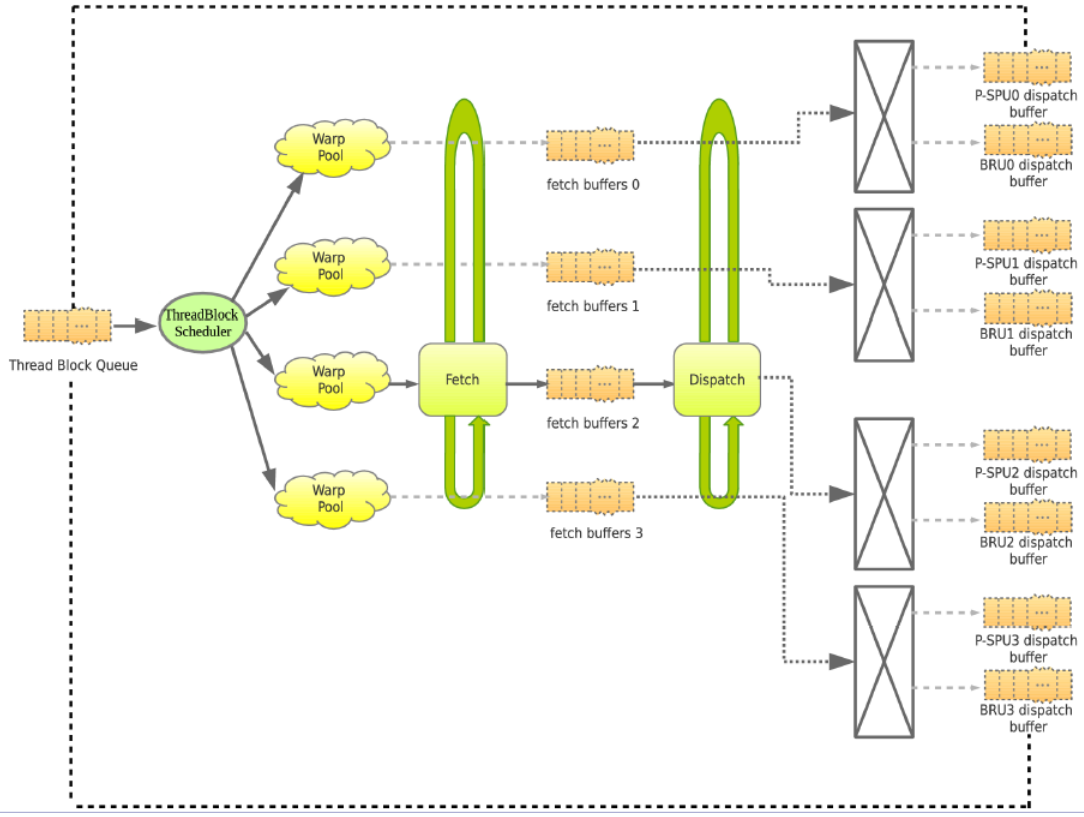

Figure 2.9: Kepler's front-end architecture

Execution Units

Execution Units have five stages: decode, read, execute, write, and complete. The execution units perform the arithmetic-logic instructions depending on data-type, and access register files in the read and write stages. The LS units are illustrated in the below figure 2.10 .

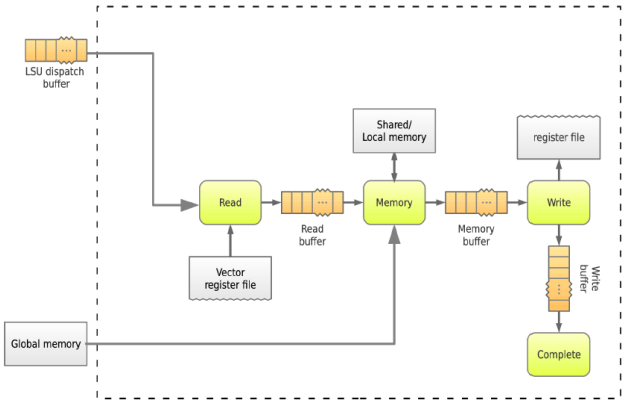

Figure 2.10: LS unit architecture

2.2.4 Operating modes

Functional Simulation

Within the first operating mode, the functional simulation, **Thread blocks** are run one at a time; through emulating instructions and updating registers and memory. There are some limitations for this simulation, as it only shows the number of executed **grids** and **blocks** and the dynamic instruction mix of the kernel.

Architectural Simulation

Secondly, the Architectural Simulation starts by modeling the **SMs** and the memory hierarchy. Then, it runs **Thread blocks** into **SMs** and **warp pools**, and, finally, it emulates instructions and delivers state within the execution pipelines. This type of simulation is for the benefit of resource modeling usage, contention , and timing accuracy.

2.3 Micro-architectural Level Fault Injector

2.3.1 SIFI

This Fault injector used in this thesis is inspired by Southern island fault injector (**SIFI**), which was designed for AMD **GPUs**, However we will evaluate the reliability of the **Kepler micro-architecture**. **SIFI** was designed originally for the AMD micro-architecture and was constructed on the top of **Multi2sim** simulator. **SIFI** provides the chance to be extended to other **micro-architectures** such as **NVIDIA Kepler**, which is our area of interest.[7]

SIFI Architecture and Functionalities

SIFIs' main focus is on the **Soft Errors** mentioned in section 1.3 within the memory arrays of **GPUs** as the Vector register file, the scalar register file, and the local memory. **SIFI** directs the **Single Event Upset (SEU)**, which is a single bit-flip of a memory element. The reliability of **GPU** is calculated by the Architectural Vulnerability Factor **(AVF)** of the addressed hardware. **AVF** scales the soft-error responsible for a system failure as Detected Unrecoverable error **(DUE)**. [5]

AVF Util is the exposure of the system for a system failure caused by a soft error occurring in a resource which is used at least once in the context of the computation. and can be calculated in terms of **AVF**, and Occupancy which represents the ratio between util resources and the total number of resources. 2.1

$$AVF = AVF_{Util} \times \ Occupancy \tag{2.1}$$

Failure In Time **(FIT)** rate of the system λ_S, it can be derived from **AVF** computed and Vulnerability of memory elements in the **GPU**:

$$\lambda_S = \sum_{i \in vRF, sRF, LM} AVF_i \ \times \ \lambda \ \times \ \#bit_i \tag{2.2}$$

In equation 2.2, $\#bit_i$ represents the number of memory elements in HW while i and λ is the error rate bit of the targeted technology node.
To calculate system performance Executions Per Failure (**EPF**) 2.3 is introduced in terms of Executions in Time (**EIT**) in 10^9 hours. **SIFI** enables the calculation of all the above-mentioned metrics.

$$EBF \ = \ EIT \ \setminus \lambda_S \tag{2.3}$$

The Fault Injection Engine

The Fault Injection(**FI**) engine allows for precise reliability analysis. The **FI** is a complex task because it requires the running of a significant number of executions. In fact, it simulates only one fault per execution, after which the output is compared with one golden execution, to break down into two categories: masked or non-masked. Non-masked faults can be filtered into **SDC** and **DUE**.

FI campaign consists of several steps. At first, the application is profiled in order to identify the time intervals in which the GPU is active and to collect information about the executed kernels. The faults to be injected are then randomly generated

and another simulation is run to profile whether these faults affect at least one hardware structure assigned to a work-groups. In case a fault hits a non-assigned hardware structure, it is marked as masked without performing any simulation. Otherwise, it is marked as *Util*. Eventually, all faults marked as *Util* are simulated and classified. Using the results of FI simulations the AVF and AVF_{Util} of an hardware structure can be computed as [8]:

$$AVF = \frac{\#inj_{not\_masked}}{\#inj} \tag{2.4}$$

$\#inj_{not\_masked}$ represents the number of not masked faults, while $\#inj$ simulates the total number of injections.

$$AVF_{util} = \frac{\#util - inj_{not\_masked}}{\#utilinj} \tag{2.5}$$

In equation no 2.5, AVF_{util} is calculated in terms of *util* faults.
The speedup obtained by skipping non-Util simulations depends on the application and mainly on the occupancy of the hardware structures. It can be computed [8]:

$$Occ_{occupancy} = \#inj./util - inj \tag{2.6}$$

2.3.2 Combining Cluster Sampling and ACE analysis

Architectural Correct Execution (ACE) Analysis

In order to identify a memory element as **ACE**, the fault must be not only masked but read too. Thus, to be considered, an element as **ACE util**, must be after reading cycle, defining what is called vulnerable timing windows (**VTW**) of a resource. Figure 2.11 shows the criteria applied.[9]

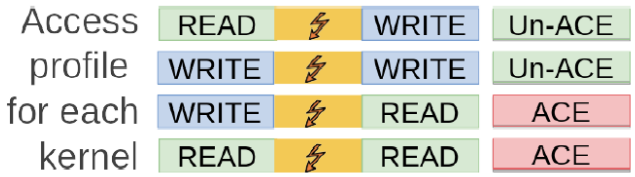

Figure 2.11: The vulnerable timing windows considered in ACE analysis

$$AVF_{Util} = AVF_{ACEUtil} \times ACEUtil_{Factor} \tag{2.7}$$

$$\frac{\bar{M}_{util}}{Inj_{Util}} = \frac{\bar{M}_{ACEUtil}}{Inj_{ACEUtil}} \times ACEUtil_{Factor} \tag{2.8}$$

In equations 2.9 and 2.10, **AVF** was calculated by *ACE util* injections, by introducing an $ACEUtil_{Factor}$ representing the ratio between util injections and ACE util injections. Following the procedure and forms used in equations 2.9 and 2.10, we obtain $\bar{M}_{Util} = \bar{M}_{ACEUtil}$ and $Inj_{ACEUtil} = Inj_{util} \times ACEUtil_{factor}$. Where $0 \leq ACEUtil_{factor} \leq 1$, simulating only $InjaceAce_{Util}$ will decrease the number of injections by a factor of $1/ACEUtil_{Facror}$.

Fault Pruning

Combining both FI and Ace Analysis results in the Fault pruning.In order to addresses the vulnerable HW resources more efficiently with respect to evaluation time and accuracy. The *fault pruning* in figure 2.12 was applied associated with *cluster fault sampling* techniques[9], In addition to the **ACE** analysis which addresses the *util resources* mentioned in **SIFI**. Fault injection is refined by identifying **ACE** resources inside util resources, as shown in figure 2.12.

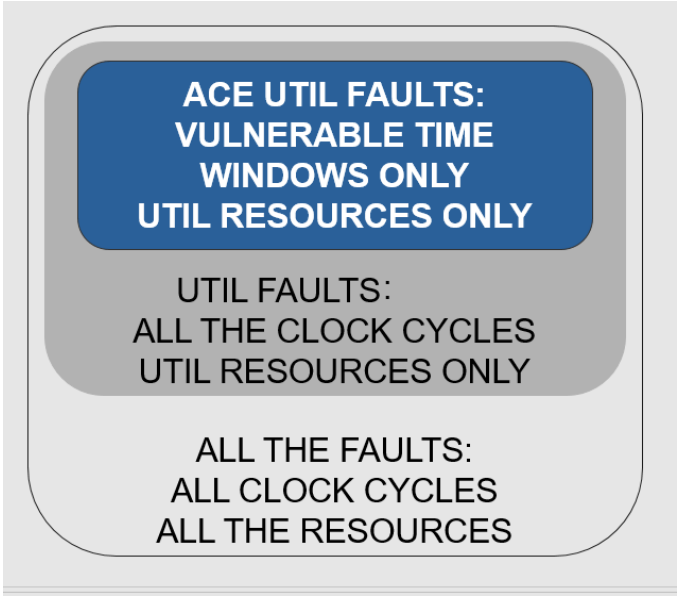

Figure 2.12: Ace Util faults

AVF is computed by redefining equations no 2.4,2.5 and 2.6, with respect to $\bar{M}$, which is the number of non-masked injections for both util and non-util injections.

$$AVF = AVF_{Util} \times Occ \tag{2.9}$$

$$\frac{\bar{M}}{I_{nj}} = \frac{\bar{M}}{Inj_Util} \times Occ \tag{2.10}$$

$\bar{M}_{util}$ represents the number of non-masked *util* injection, I_{nj} is the number of both util and non-util injections, and In_{jutil} represents only the number of util injection. Therefore, $In_{util} = Inj \times occ$, considering that $\bar{M} = \bar{M}_{util}$ as the $non-util$ injections are always masked.

Finally, simulating only In_{jutil} and knowing that $0 \leq Occ \leq 1$, the number of injections with a factor of $1/Occ$ diminish.

Injection Sampling

The cluster sampling occurs in two steps: a first step consists of sampling the clusters, and the second is identifying the individuals from the selected clusters. The **VTW** is considered a cluster, and its weight is modeled by its duration (Clock cycles).
In this thesis, **Wight and Sample (WAS)** technique is used. the first sampling stage is based on proportional to size sampling **(PSS).** Clusters are selected with a probability proportional to the associated w_i. Once the clusters are selected an injection is evaluated for each of them. The second sampling stage is based on uniform sampling and the same number of individuals must be analyzed for each of the selected cluster. In this particular case we consider just a single individual per cluster as its outcome is the same to the other individuals in its cluster. If the outcome is non-masked, then $ai = 1$, otherwise $ai = 0$. With this approach, adapting the theory introduced in 2.4, 2.5, and 2.6 to our case, the $AVF_{ACEUtil}$ [9] can be:

$$AVF_{ACEUtil} = \frac{\sum_{i=1}^{n} \alpha_i}{n} \tag{2.11}$$

with a standard error equal to

$$se(AVF_{ACEUtil}) \sqrt{\frac{\sum_{i=1}^{n} (\alpha_i - AVF_{ACEUtil})^2}{n(n-1)}} \tag{2.12}$$

2.3.3 The Proposed Workflow

The workflow adopted to estimate reliability consists of several steps:

1. Application profiling
   - Identify VTW concerning:
     - duration and first clock cycle
     - involved micro-architectural regiters file
   - Collection information about kernels execution

– parameters needed to map architectural registers to the physical one

2. Fault pool is generated using WAS
3. Faults are injected
4. Faults are classified according to PSS

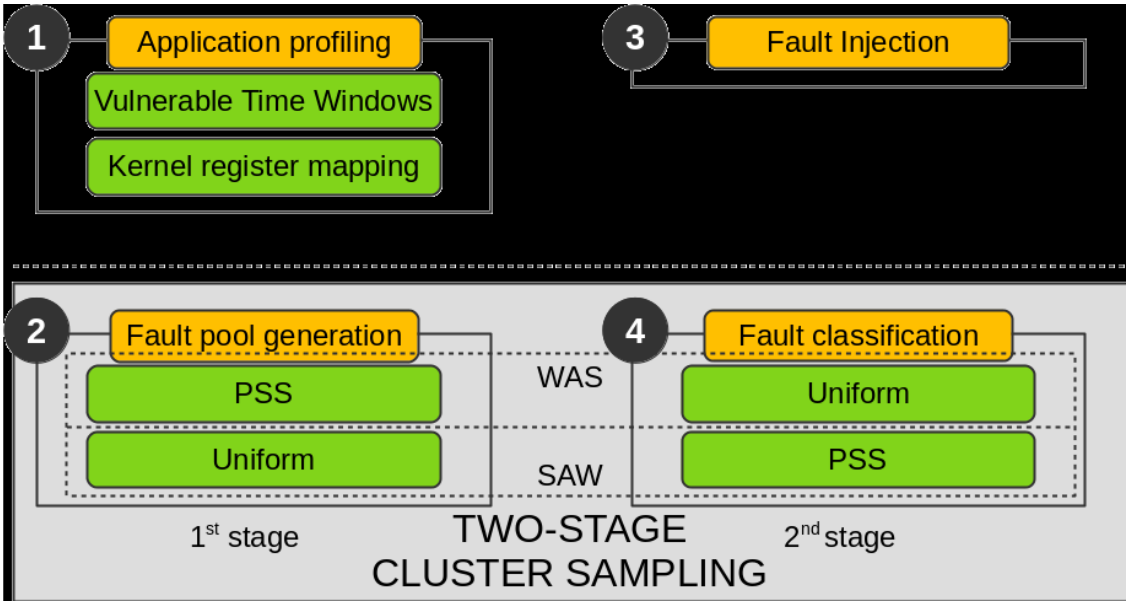

Figure 2.13: The proposed Workflow

Chapter 3

Experimental results

We adopted the workflow presented in chapter **2.3**. We simulated **1000** Injections of single bit upset in the register file and used four benchmarks of **CUDA**: VectorADD, VectorADD-Int, MartixMul, and ScalarProd.

First, the **VTW** was identified using **ACE analysis**. Second, the fault pool was generated using **WAS**. Finally, faults were injected and classified according to **PSS**. The output of the workflow is shown in table 3.1 and figure 3.1.

	VectorADD	VectorADD-Int	MartixMul	ScalarProd
SDC	880	906	899	283
DUE	0	0	59	220
Masked	120	94	42	497
TOT	1000	1000	1000	1000

Table 3.1: Output of the simulation

SDC was encountered the faulty simulation output mismatched from a golden simulation. The **DUE** was detected when when the simulation crashed or the elapsed time of the running simulation was 5x longer than the one of the golden simulation.

According to A. Vallero and S.Di Carlo. [2] the *confidence interval* calculated is 95% with an error $e = 3\%$

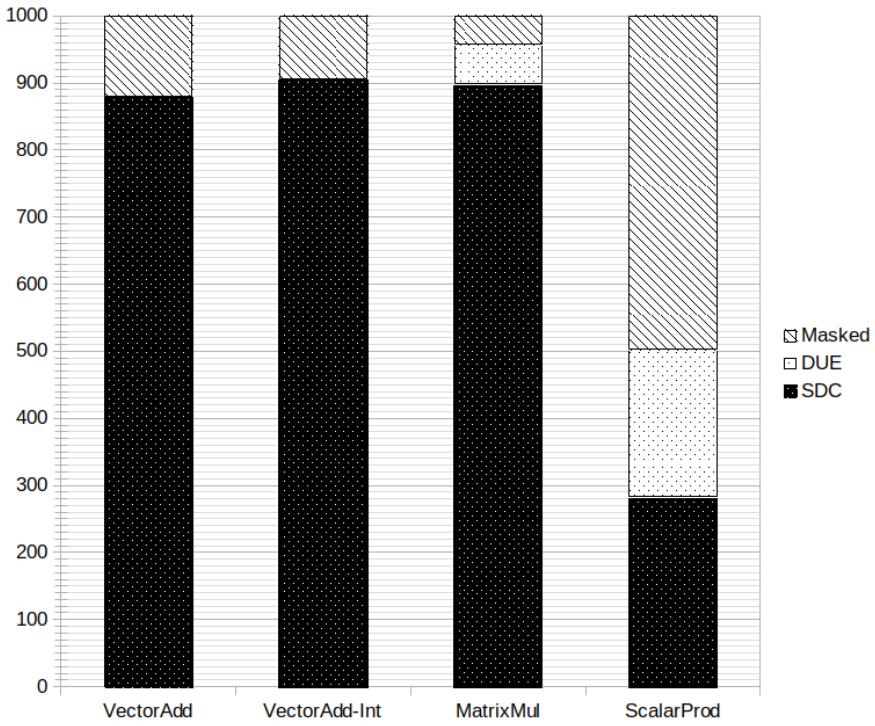

Figure 3.1: Simulation Output

Chapter 4

Conclusion

Kepler-Multi2sim is an impactful tool. By allowing fast and efficient simulation, it enables the modification of the running **micro-architecture**. However, the findings of this study show that it also has some drawbacks. One drawback is that the memory hierarchy of the timing simulation displays some defects. Second, some unmodeled instructions were detected while using the Rodinia benchmark. Addressing these issues may constitute a starting point for future work on the subject.

The 2 stages cluster sampling strategy demonstrated a good fit for fault pruning with respect to minimizing the number of injections. The adopted analysis allows better reliability and faster time for **GPGPU** applications to access the market. Due to limitations of this analysis, we adopted only the **WAS** methodology which sharpened our simulations, while other methods such as **SAW** could be utilized in future studies.

Bibliography

[1] M. Arora et al. “Redefining the Role of the CPU in the Era of CPU-GPU Integration”. In: *IEEE Micro* 32.6 (Nov. 2012), pp. 4–16. DOI: 10.1109/MM.2012.57.

[2] A. Biswas et al. “Computing Accurate AVFs using ACE Analysis on Performance Models: A Rebuttal”. In: *IEEE Computer Architecture Letters* 7.1 (Jan. 2008), pp. 21–24. DOI: 10.1109/L-CA.2007.19.

[3] X. Gong, R. Ubal, and D. Kaeli. *Multi2Sim Kepler: A detailed architectural GPU simulator*. Apr. 2017, pp. 269–278. DOI: 10.1109/ISPASS.2017.7975298.

[4] E. Lindholm et al. “NVIDIA Tesla: A Unified Graphics and Computing Architecture”. In: *IEEE Micro* 28.2 (Mar. 2008), pp. 39–55. DOI: 10.1109/MM.2008.31.

[5] S. S. Mukherjee et al. “A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor”. In: (Dec. 2003), pp. 29–40. DOI: 10.1109/MICRO.2003.1253181.

[6] R. Ubal et al. *Multi2Sim: A simulation framework for CPU-GPU computing*. Sept. 2012, pp. 335–344.

[7] A. Vallero, D. Gizopoulos, and S. Di Carlo. “SIFI: AMD southern islands GPU microarchitectural level fault injector”. In: (July 2017), pp. 138–144. DOI: 10.1109/IOLTS.2017.8046209.

[8] A. Vallero, D. Gizopoulos, and S. Di Carlo. “SIFI: AMD southern islands GPU microarchitectural level fault injector”. In: *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*. July 2017, pp. 138–144. DOI: 10.1109/IOLTS.2017.8046209.

[9] A. Vallero and S.Di Carlo. “Combinig cluster sampling and ACE analysis to improve fault-injection based reliability evaluation of GPU-based systems”. 2019.

This Ph.D. thesis has been typeset by means of the TEX-system facilities. The typesetting engine was pdfLATEX. The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete TEX-system installation.