An extended RT-level model of an NVIDIA GPU for safety-critical applications

Master Degree in Electronics Engineering

Supervisors:

Matteo Sonza Reorda

Josie Esteban Rodriguez Condia

Luca Sterpone

Author:

Riccardo Faggiano

April, 2022

Acknowledgement

I would like to start by thanking Professor Matteo Sonza Reorda for accepting my application and for believing in me from the beginning.

A special thanks goes to Dr. Josie Rodriguez for supporting me throughout this long journey and for always proving to be a good person. Without him all this would not have been possible and I wish him the best in life because he deserves it.

I thank my father and my sister Marta for always being by my side, for supporting me in all my choices and for giving me the strength to go on.

A special mention to all my friends, for all the laughs they gave me and for helping me to overcome the most difficult moments.

To all the people I met during this experience who have a place in my heart.

Finally, I want to thank my mother. My angel who protects me from heaven. To you I dedicate this and all that I will reach in life. The best part of achieving a goal was seeing happiness in your eyes. My battles were yours too. Until the last day you have always dedicated your life to your family. I hope you are well and proud of what I am doing. I love you so much and miss you so much.

Abstract

A General Purpose Graphics Processing Unit (GPGPU) is a Graphic Processing Unit (GPU) that is programmed for purposes beyond graphics processing, such as performing computations typically performed by a Central Processing Unit (CPU). These architectures are focused on the exploitation of parallelism through the execution of many processes at the same time on different hardware units. The latency is not the only metric considered for the GPU, while instead the throughput becomes fundamental for the evaluation of the performances. Adopting a GPU for general purposes allows the CPU architecture to accelerate portions of an application and this leads to high performance and faster programs. They have been massively used in the last years in many fields, such as DSP, bioinformatics, machine learning, etc. They are becoming popular also in safety-critical applications as, for example, autonomous and semi-autonomous vehicles.

These devices suffer from transient faults because of their high structural density. Among these faults, it is possible to consider SEUs (Single Event Upsets) which are changes of the state of a transistor in a node of an electronic device produced by radiation effects. Their presence is very dangerous in micro-systems and it is not acceptable in critical applications because it can cause misbehaviours. For this reason, it is fundamental to analyse the consequences of faults on internal blocks of these devices to certify the products for the industrial market.

The scheduler controllers are internal modules which have an extreme importance in the management of the applications running on the GPU: they are employed especially during the configuration phase and are used to dispatch the job workload among the various units of the hardware. Furthermore, the evaluation of the fault sensitivity can be a way to study and develop proper mitigation mechanisms.

The impact of the transient faults in the GPU devices can be analyzed in different ways. One strategy to deal with it is based on the model simulation. By the way, following this path requires a complete model which can provide in detail the low-level characteristics of the internal module to study. Unfortunately, the low-level micro-architectural reliability characterization of the Block Scheduler received limited attention, mainly due to the lack of appropriate GPU models and to their complexity.

The aim of this work is to evaluate the impact of transient faults on the general controller of a GPU. For this purpose, the capabilities and description of a low-level micro-architectural GPU model (*FlexGripPlus*) are extended to support the management of several execution cores (i.e., Streaming Multiprocessors) and allow the analysis of fault effects.

The general scheduler controller has a crucial role in the structure and it has been extended to work with many SMs. So the design phase has started with the extension of this module. The Round Robin algorithm adopted in the FlexGripPlus for scheduling purposes has been modified to perform its job in the presence of many cores. The presence of a generic number of Streaming Multiprocessors also has to do with all the memories: the access to them must be accurately managed to avoid contention issues. So, first of all, the structure has been modified both inside the SM and outside. Some modules have been added, such as the arbiters, to avoid eventual contention and to make possible the right access of shared resources. Internal structures have been adapted to allow the correct synchronization and some signals are added to control this mechanism. At the end, the arbiters are internally redesigned to perform their action correctly with a variable number of SMs.

Each design modification has been followed by an intense simulation phase to prove the correctness of the model. The results were compared with the ones obtained in the original FlexGripPlus model with only one SM. Design errors were checked and corrected through the observation of the signals in the ModelSim tool. Different applications with different configurations have been simulated to cover

as many cases as possible.
A set of fault simulation campaigns were performed to evaluate the incidence of transient faults in the applications. Faults lists are generated and transient faults are injected to analyze the effects on the general controller. The experiments have been conducted on a different number of applications which have been configured in various ways (changing, for example, the number of SMs and the workload to be performed) to cover and study different cases.
The results show that, for some signals in the general controller, the faults are propagated mainly during the configuration phase and the dispatching of the workload. Furthermore, for certain configurations the device with 2 SMs is more reliable with respect to the version with only one SM. We also observed that the distribution of the program (so in terms of blocks) is fundamental and it can affect the vulnerability of the block scheduler.

Contents

List of Figures

List of Tables

1 Introduction

Graphic Processing Units (GPUs) are processors mainly focused on the parallel execution of the application involving several units to perform the same job at the same time. Their highly parallel structure makes them very suited for algorithms and programs where high amount of data are involved. They are commonly used together with a CPU which can instead be exploited for applications where the performance in terms of throughput and computation of large data is not essential. Nowadays, GPUs are used also for general-purposes, so we talk about General-Purpose Graphic Processing Units or GPGPUs. They have been massively used in the last years in many fields, such as in sensor-fusion, computer vision [1], DSP [2], bioinformatics [3], machine learning [4]. They are becoming popular also in safety-critical applications as, for example, autonomous and semi-autonomous vehicles [5, 6] and mission-critical (i.e., autonomous systems [7]).

Some studies have proven that these devices suffer from internal effects (such as aging and wear-out) and from external ones (i.e., Electromagnetic Interference EMI or radiation effects) [8]. SEUs (Single Event Upsets), for example, belong to the second category and they constitute a change in the state of transistors caused by electromagnetic effects. It is not acceptable to not be reliable if the GPUs must be empowered in safety-critical applications otherwise the consequences could be catastrophic. For all these reasons, reliability analysis in GPUs is crucial when developing products for the safety-critical domains.

The reliability analysis can be performed in 3 different ways mainly: beam experiments, software-based fault injection and simulation-based fault injection. In the first approach, the devices are placed in environments in order to be exposed to electromagnetic effects, such as radiations, to facilitate the creation of faults. The problem is that is not easy to understand the relationship between the effects of the faults and their origin in the internal blocks. The second strategy, software-based fault simulation, is based on error models to represent hardware faults. Even in this case, the analyses are not performed accurately because of lack of observability and controllability of functional units. The last way, the simulation-based fault injection approach, is based on the usage of low-level GPU models (for example at RT o gate level) which are very suitable for these kind of fault analysis for their controllability, flexibility and observability. Due to the lack of appropriate GPU models, the reliability characterization of some internal modules, such as the Block Scheduler Controller, has not been properly performed.

For this reason, the purpose of this work is to extend an open-source low-level micro-architectural model, called *"FlexGripPlus"*, to handle a multi-cores systems design. In this way, a reliability analysis on the block scheduler can be performed considering transient faults (Single Event Upset) injection and different applications with various configurations of cores and workloads.

The current thesis report is organized as follows: in the first chapter a brief introduction about GPUs and the description of the FlexGripPlus model are given; the second chapter describes the design flow followed to reach an architecture with a generic number of SMs, first in a general way and then focusing in detail on all the main HW modifications; a chapter related to the experimental results follows: it contains the outcomes of the validation phase and those ones of the fault injection campaign; in the end, a conclusive part of the job is present as final chapter.

2 GPU background

Until the mid-2000's, computer designers used to increase the clock speed to get better performances. The first personal computer (in the 1980s) had a internal clock speed around 1 MHz. Nowadays, most CPUs have a clock speed between 1 GHz and 4 GHz, which means nearly 1,000 times faster than the original processor [9].
This strategy was no more advantageous beyond certain limits due to heat and power consumption issues. Thus, engineers decided to leave the traditional single-core architecture to explore the world of *"parallel processing"*: this means investing in the parallelism through the exploitation of multiple hardware (multi-core systems) to perform a big number of computations at the same time.
GPUs or *Graphic Processing Units* were the first architectures following this mechanism and they were employed especially for the Graphic Processing in the gaming market. They were commonly based on the Single-Instruction, Multiple-Data (SIMD) taxonomy [10]: a single control unit which takes a single instruction with multiple execution units in parallel.

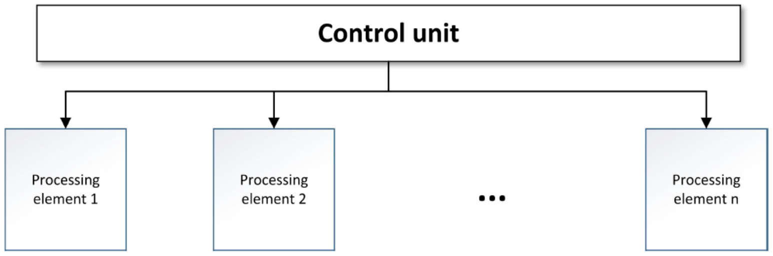

Figure 1: SIMD architecture, extracted from [11]

GPUs are special-purpose processors mainly developed to provide a high throughput by exploiting hardware parallelism when executing an application. In a GPU, the area is mainly devoted to the data processing units while a small part is dedicated to data caching and flow control unit: this makes them very well suited for algorithms where a high amount of data must be processed. On the other hand, CPUs are more indicated for control flow intensive programs [12].

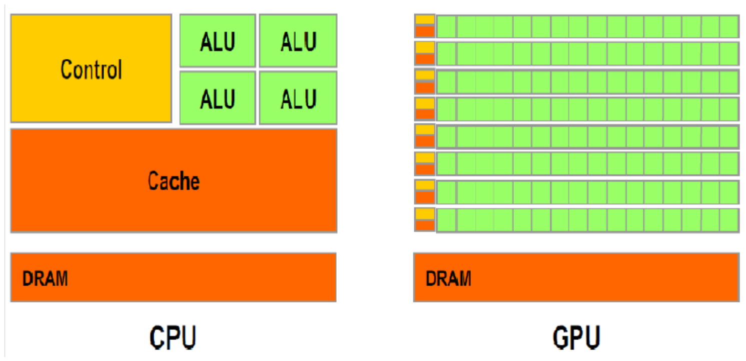

Figure 2: Differences between GPU and CPU, extracted from [12]

In the last decades, *General-Purpose Grapich Processing Units* (GPGPUs) have become successful: they are Graphic Processing Units (GPUs) that are programmed for purposes beyond graphics processing.
In this work, a low-level micro-architectural GPGPU model (*"FlexGripPlus"*) is considered for our purposes.
The next sections are dedicated to the general description of the GPU, starting from a high level programmer view and then analyzing the same concepts at low-level in terms of Hardware. Then, the FlexGripPlus model is presented.

2.0.1 GPU programming model

The programming model in a GPU provides the main methods and strategies to use the hardware resources in the GPU devices. Thus, the schedulers are also configured according to the programming model and the descriptions in a parallel program.
The programmer establishes the configuration to run in a certain application: he decides the number of cores, the number of blocks to be executed in each SM (which means it decides the workload at each execution of the program) and the number of threads. All these concepts are useful also on the hardware side. So in this way, a complete view of the GPU is given and this facilitates the understanding of the structure explained in the next sections.
Programming a GPU is certainly different from programming a CPU. The main idea is to understand that you will write a piece of code as in normal computer programming but that it will be executed many times in parallel by the device.
Different methods exist for programming a GPU: CUDA, OpenCL and other high level libraries such as OpenACC and Trust. For the purpose of this work, just the *CUDA platform* (Computer Unified Device Architecture) is considered: it is generated by NVIDIA and the C and C++ programming languages are extended to support parallel programming for GPUs.
In a computer system, the GPU is a *"device"* which needs a *"host"* in order to start working. The CUDA programming model allows to program both processors with just one code. Part of the CUDA program is written in normal C and it will run on the host, while the remaining part, written with

extension for parallel programming, will run on the GPU.

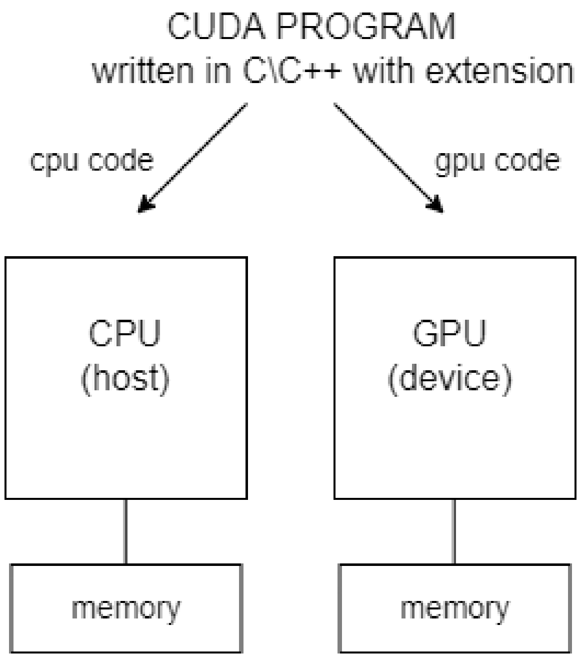

Figure 3: CUDA program, adapted from [13]

CUDA assumes that the device, the GPU, is a co-processor to the host, the CPU. It also assumes that both the CPU and the GPU have their own separate memories where they store data.
The CPU is in charge of different tasks: it runs the main program and it sends directions to the GPU to tell it what to do. It is the part of systems responsible for the following:

- move data from CPU to GPU;
- move data from GPU to CPU;
- allocate GPU memory;
- launch kernel on GPU.

The *Kernel* is a function executed in parallel on the device. So, a typical program on the GPU does these operations in order:

1. CPU allocates storage on GPU;
2. CPU copies input data from CPU to GPU;
3. CPU launches kernels on GPU to process the data;
4. CPU copies results back to CPU from GPU.

The kernel can be executed by a variable number of *threads*. Threads are grouped in *blocks*. Blocks are grouped in a *Grid*. Then, within a block of threads, the same threads are executed in groups of 32, called *warps*. The programmer must configure the kernel defining the *"grid of blocks"* (in 1, 2 or 3 dimensions) and the number of *"threads"* per each block (again in 1, 2 or 3 dimensions):

$$kernel <<< GRID\ OF\ BLOCKS, BLOCK\ OF\ THREADS >>> (...)$$

The organization of the grid of is crucial in the definition of the operations performed by the general controller at low-level. In fact, depending on the dimensions of the grid along all the axis, the block scheduler can perform a different numbers of scheduling activities. Obviously, the workload of the block scheduler also depends on other factors as, for example, the thread occupancy of the SMs.

2.0.2 GPU Hardware

A GPU is composed of an array of parallel processors, called *Streaming Multiprocessors* or *SMs*. Each of these is organized in 5 pipeline stages, which are *Fetch*, *Decode*, *Read*, *Execute* and *Write*. Each of these stages has a defined role in the execution of the instructions. In the Execute stage, several *Scalar Processors* (*SPs*) are present and they are the functional units in charge of performing the computation needed for each thread. Other important elements in the SM are the memories: you can find the register file, the shared memory and the local memory. These are all placed inside the SM device, but outside the cores it is possible to find other kinds of storage elements such as the global memory, the constant memory and the system memory, each with a defined role.
Modern GPU architectures may also include accelerators implementing complex operations, such as the cross product of matrices (*Tensor cores*), floating point operations (*FP32 or FP64*) and trascendental functions (*SFU*) [8].
In the application execution, a crucial role is played by the schedulers. They are fundamental for the management operation and the division of the workload among the various cores. There is a general controller, the *Block Scheduler*, outside the SMs which is used to schedule, following precise distribution policies, the blocks among the Streaming Multiprocessors. After that, another scheduler is present inside the SM, the *Warp Scheduler*, which dispatches warps into the available SPs for the execution.
The figure 4 shows a general scheme of a GPGPU device with 4 SMs:

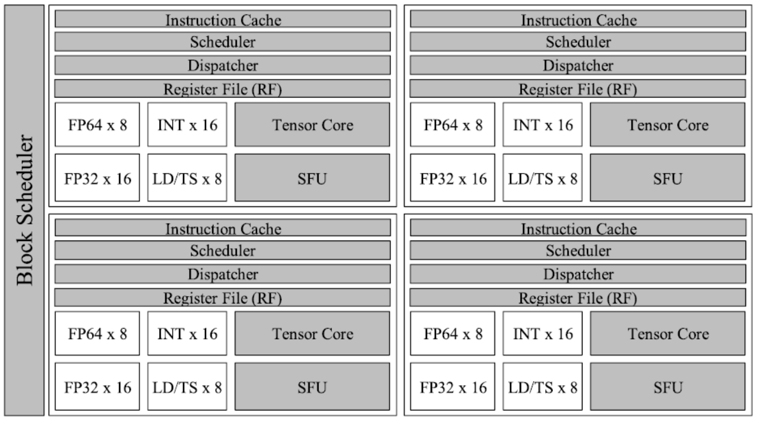

Figure 4: General scheme of a GPGPU, extracted from [8]

A GPU from both software and hardware point of view is represented in Figure 5:

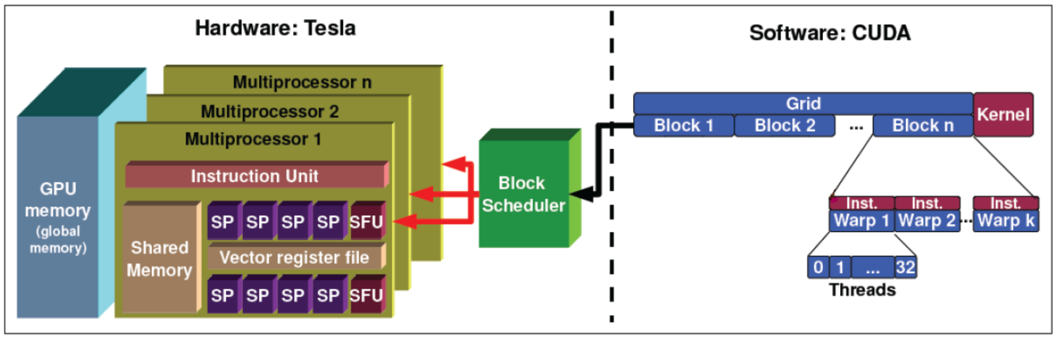

Figure 5: HW and SW GPU environment, extracted from [12]

On the software side, it is possible to observe that the kernel is executed by a grid of blocks, where each block is divided in several warps and the warps are groups of 32 threads. On the hardware side, you can observe that the GPU is composed of several Streaming Multiprocessors, each composed of the functional units and memories previously mentioned. Then, you can notice that the main memory is represented outside the SMs.
In this picture, the role of the block scheduler, main controller in the GPU, is highlighted. It is the link between the software and hardware world in the architecture. It is crucial in the management of the hardware resources and in the execution of a parallel program. For this reason, the reliability analysis on this module is so important: in fact, a fault in this unit can compromise the operation of the whole device.

2.1 FlexGripPlus Architecture

"FlexGripPlus" or *"FLEXible GRaphIc Processor Plus"* is an open-source soft-GPGPU model based on the NVIDIA G80 microarchitecture and implemented in VHDL.

It is an extended version of the *FlexGrip* model: it was originally developed by the University of Massachusetts Amherst and was designed to be fully compatible with the CUDA programming environment using the SM 1.0 compatibility [14]. FlexGripPlus was developed by Politecenico di Torino on top of FlexGrip and it overcomes some limitations of the previous version: the dependency on the Xilinx FPGA library, the limited number of supported instructions and the incomplete compliance with the NVIDIA application development environment. By the way, during the FlexGripPlus development, the micro-architecture of the original model has been kept untouched [8].
The FlexGripPlus model supports only one SM mainly composed of a five-stages pipeline (*Fetch*, *Decode*, *Read*, *Execute* and *Write-Back*). The flexibility of the model allows the selection of 8, 16 or 32 SPs.

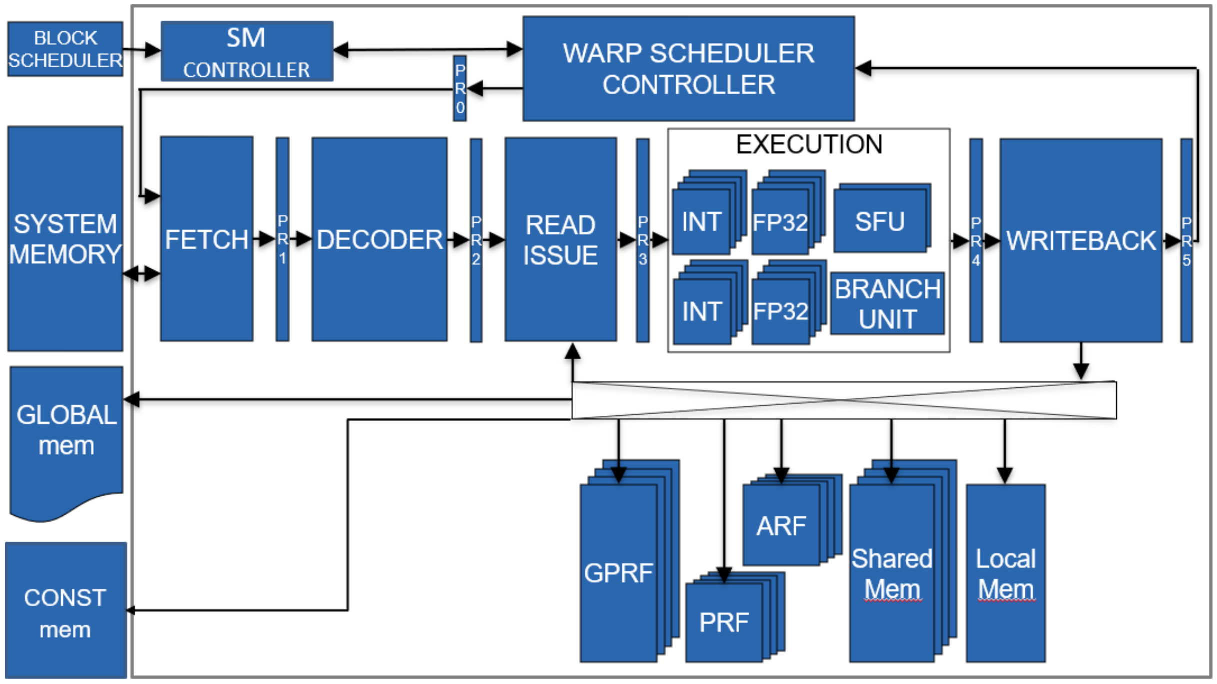

Figure 6: General scheme of the SM in the FlexGripPlus model, adapted from [15]

2.1.1 Pipeline stages

This architecture is composed by 5 pipeline stages. There is no central control unit because each stage communicates to the following one when it is time to start working [16].
The *Fetch Stage* is in charge of fetching instructions from the memory and passing them to the decode stage. The instruction bus is 32 bit-wide while the instructions could be on 32-bit (half instruction) or 64-bit (full instruction): this means that an instruction can be fetched in 1 or 2 clock cycles depending on their length.
The *Decode Stage* is a very complex module in charge of decoding the instructions based on different fields (opcode, size of the operands, conditional fields, etc.) to understand what is the operation to

perform in the next stages.
The *Read Stage* has 3 internal modules which work simultaneously and are able to retrieve in parallel the operands from the vector register file and serially from the other memories [17].
The *Execution Stage* is one of the most important parts of the entire architecture. In this unit, the scalar processors are present inside. Each of them executes one single thread. The SPs perform arithmetic operations such as multiplication, addition, subtraction, data type conversion and boolean logic and shifting operations. In this stage also the logic for the control flow is present which is widely used for the branches and for all the mechanism of threads synchronization [18].
The *Write Stage* is the last stage of the pipeline. It writes back the results of the execution in the memories.

2.1.2 Memory organization

GPUs are parallel architectures and are mainly based on big arrays of parallel processing cores. However, each core requires data operands to process. Thus, memories are needed in the structure to allow load and store operations for each thread executed. In these parallel architectures, the massive number of active threads per applications forces the adoption of several levels of memory resources to hide latency when the instructions are executed [15].
It is possible to do a distinction between *"In-chip"* and *"Out-chip"* memories: the first are located inside the SM while the second outside. *"In-chip"* memories are composed of *Register File*, *Predicate Register File*, *Address Register File*, *Local memory* and the *Shared Memory*. Instead, the *Global memory*, *Constant Memory* and *System memory* belongs to the group of *"out-chip"* memory.
The *Vector Register file* is the fastest in-chip memory resource and it is used to store immediate results. Each thread, which is mapped to a precise SP, has its own portion in this kind of storage. All threads can access the Vector Register File but no one threads can access the portion of memory not reserved for it.
The *Predicate Register File* stores the predicate flags per thread as result of logic, arithmetic and setting instructions. These flags are also used in conditional instruction operations.
The *Address Register File* contains registers only used for addressing the shared memory in an indirect way (that means with an offset).
The communication among threads is achieved trough the *Shared Memory* which is obviously inside the SM. The Shared Memory contains information about the blocks assigned to the multiprocessor and each block has its reserved portion in it.
The *Local Memory* is a small memory used to store array-type data commonly employed in a parallel program.
The *Global Memory* is visible by all the SMs and stores inputs and outputs of the kernel for the communication with the host. The *Constant Memory* is a read-only space where the kernel constants are written by the host and the *System Memory* contains the instructions of the kernel. These last three kind of memories are all outside the Streaming Multiprocessor [17].

Talking about memories, it's worth introducing the *"arbiter"*. It is a very common block used in HW design when a common resource is wanted by several users. The arbiter is the piece of Hardware who decides who gets to use the common resource at any given time [19].
Arbiters may be realized in software or in hardware. Software arbiters normally require many machine cycles to resolve an access request whereas an hardware arbiter can be designed to operate in one clock cycle (or less). Arbiters may also be classified in centralized and decentralized [20].

The arbiter present in the design is characterized by this kind of entity:

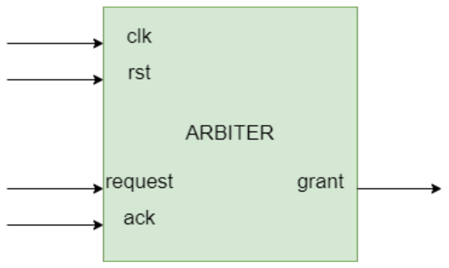

Figure 7: Arbiter entity

Each signal has its defined role:

- **request**: asserted by each one of the devices that want to own the common resource;
- **grant**: asserted by the arbiter to select the resource which will use the service;
- **ack**: asserted by the device which has finished to use the common resource;

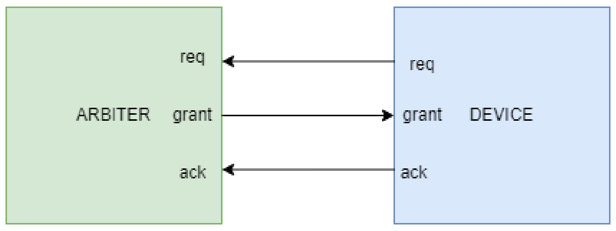

Figure 8: Handshake mechanism between arbiter and a general device

It is a *"round robin"* arbiter: it tries to serve all the resources equally. This means that there is a priority mechanism. If more requests arrive at the same moment, it will serve only one of the devices requesting the common resource (so all the other requests are lost) and then it will wait for new requests. The next time this situation will be repeated, the device already served cannot be served again and another one will be considered for the selection.

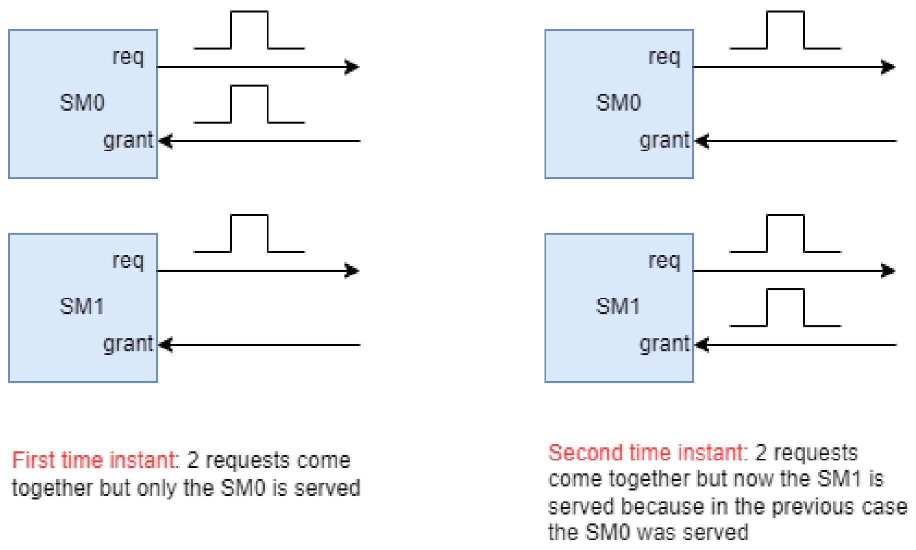
req
SM0
grant
req
SM1
grant
First time instant: 2 requests come together but only the SM0 is served
req
SM0
grant
req
SM1
grant
Second time instant: 2 requests come together but now the SM1 is served because in the previous case the SM0 was served

Figure 9: Multiple contemporary requests management

In the FlexGripPlus design, different arbiters are used, especially locally in the SM. In fact, for example, an arbiter is used in the *"Read Stage"* to manage the requests coming from 3 elements, the *"read source operands"*, which are in charge of requesting the operands from the global memory.

2.1.3 Scheduler Controllers

The schedulers in a GPU are devoted to organize and manage the execution of an assigned task into the available resources.
The first type of controller is the *Block Scheduler Controller* (also known as *Task Scheduler Controller*): it is the module in charge of associating each block of the grid to the single SM in the FlexGripPlus design. Its general operation starts after the GPU configuration. So, at this point, it dispatches the blocks considering the maximum block occupancy of the core. Some of the blocks composing the grid are scheduled and they are the *"active"* ones. The others, which can't be scheduled in this phase, are the so called *"waiting"* blocks. As soon the execution of the first group is terminated, the control is passed again to the block scheduler which selects other blocks among the waiting ones to be scheduled on the SM. This mechanism is repeated until all the blocks are dispatched on the SM. This means that the block scheduler controller is active mainly during the initial configuration phase and during the moments when the active blocks and the waiting blocks are exchanged.
The block scheduler receives some configuration signals by the *"gpgpu configuration block"* which contain some information about the shared memory size, the number of blocks per each core, the dimension of the grid and so on. At the same time, the controller is connected with the streaming multiprocessor and it defines the blocks to be executed, it sends the enable signal for the starting of the operations and it gives other useful information for the application execution.
Inside the block scheduler an FSM is implemented. One of this stages is the *"CALC_BLOCKS"* which, as the name says, it is used to compute the blocks to be scheduled in the SM. The computation in the FlexGripPlus design is very easy because only a core is implemented, so this state is not very complex.

After that, the information are passed to the SM when the FSM is in the *"CHECK_BLOCKS"* state. Another state, the *"SCHEDULE_BLOCKS"* is used to update the counters for the scheduling. At the end, in the *"SCHEDULE_WAIT"* state, the block scheduler waits for the SM to finish the execution of the blocks scheduled. The Figure 10 shows the FSM state diagram:

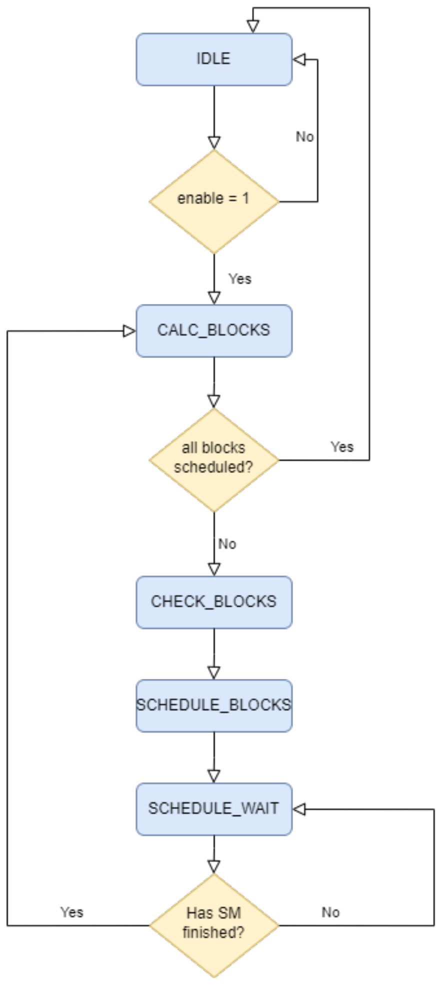

Figure 10: Block Scheduler Finite State Machine Diagram

The role of the block scheduler is crucial in the management of the application and the correct execution of the program depends mainly on it. To understand better the importance of this module, an analogy can be helpful. The GPU can be seen as a company in which the owner (in our case the programmer) decides what work to do (application) and how to do it (configuration of blocks and threads). The owner then notifies the team leader of the tasks to be performed. The head of the team (block scheduler) has the role of dividing the workload among the employees (hardware resources, i.e. the SMs). Each employee must carry out the work in parallel and must finish it within the established

time frame for the project in order to move forward. Obviously, each employee can work for a maximum of hours a day (maximum block occupancy for SM) otherwise they would not be profitable anymore. The team leader must therefore consider dividing the work into several days by assigning one part of the project at a time (exchange phase between active and inactive blocks).When an employee does not perform his duty or the team leader is not good at managing the project, the desired product cannot be obtained and the company fails (fault situation where cores does not process correctly the information).
Different algorithms exist for the purpose of scheduling blocks. In the CUDA Fermi GPU for example, three possible sequences scheduling have been identified: the *Ordered*, the *EvenOdd* and the *Random* [21]. The unique difference among the three defined sequences concerns the selection of the SM on which each block must be scheduled. The *Ordered* sequence defines the SMs in an ordered way (i.e. from the first to the last), instead the *EvenOdd* first schedules blocks to the SMs with an even SM ID and then to the one with an odd SM ID. Eventually, in the *Random* sequence SMs are randomly selected. The selected one in this case is the *Round Robin* algorithm. It is a resource allocation algorithm widely used in CPU scheduling and network packet transmission. It is simple, starvation-free and easy to implement in hardware. Blocks are assigned to SMs in equal portion and in circular order, without any priority. Although this scheduling in hardware is simple and fast, it is not efficient in GPU as it does not consider the workload characteristics of threads and the resource balance among SMs [22].
By the way, having only one SM in the FlexGripPlus design makes this algorithm not very useful.
Another controller is the *"Streaming Multiprocessor Controller"* which is the module that receives the *"go"* command from the block scheduler. Its first task is allocating the blocks assigned to the multiprocessor by dividing the shared memory according to the size indications and by writing at the beginning of each reserved portion a block header. Once initialized each block in the shared memory, the SMP controller must initializes the threads in the vector register file [17].
The control is then passed to another type of controller, the *Warp scheduler controller*: it manages the assigned blocks and splits them in Warps. It is located inside the SM unlike the Block Scheduler controller. Even in this case the round robin policy is implemented for this kind of scheduling.

3 Model extension to support reliability analysis

A bottom-up approach is the main strategy used to perform the extension of the model to reach a multi-cores system. The design process can be divided in 3 different steps:

1. Blocks distribution;
2. Memory Management;
3. Arbiter Handling.

The first task implies the modification of the block scheduler in order to be fully compatible with a design with a variable number of streaming multiprocessors. To do that, the scheduling algorithm already adopted in the original version of the FlexGripPlus architecture must be extended.
The memory Management phase, instead, is devoted to modify the internal and external structure of the SM, touching control signals and control devices, in order to manage correctly the accesses to the shared resources, as for example the global memories.
Finally, the arbiter is extended and modified to allow the correct synchronized behavior when addressing shared resources in presence of a many-cores system.
The main intention of the proposed extension was to complete the description of the FlexGripPlus model following the original micro-architectural descriptions listed in the Tesla micro-architecture by Nvidia [23]. Thus, first, the main idea was to describe in hardware the main Texture/Processor Cluster or TPC, which is composed of two SMs and then extend the GPU model to several sets of clusters.
The next subsections describe the three steps performed in the extension of the whole design.

3.1 Blocks distribution

The first step consists on updating the scheduler controller, allowing the management of several SMs and the distribution of the blocks from a parallel program. For this purpose, we modified the description inside the scheduler controller (state machine) in order to allow the distribution of blocks to more than one SM. Thus, the *"Round Robin"* algorithm implemented in the Block Scheduler has been adapted. In addition, several control lines were included to enable the operation of the available SMs and synchronize the execution of instructions after the configuration phase.
In the FlexGripPlus design, the Blocks Scheduler has an output signals vector, *"num_blocks_out"*, to define the number of blocks to be scheduled on the SM. This vector must be converted in an array of vectors when considering a multi-SM systems in order to connect the block scheduler with each of the SM. So the length of this array depends itself on the number of SMs present in the design, as shown in Figure 11:

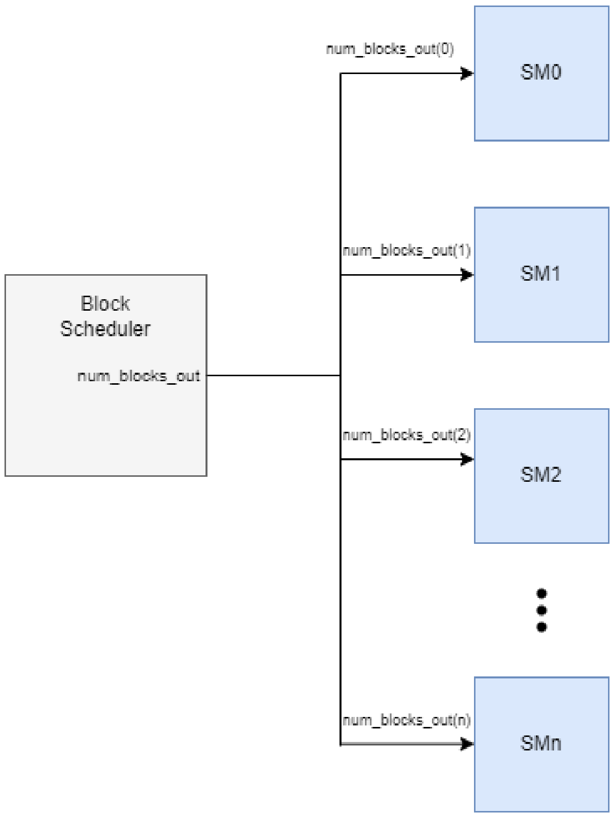

Figure 11: Blocks Scheduler output array

The Block scheduler contains a *Finite State Machine* to perform the scheduling (Figure 10 in Chapter 2). The flow of the states composing the FSM has not been changed for the extension of the model. By the way, one of the states, *"CALC_BLOCKS"*, the one dedicated to the computation of the number of blocks for each SM, has been enlarged in order to deal with a multi-core systems. The "Algorithm 1" shows the pseudo-code of the extended "Round Robin" algorithm performed in the *"CALC_BLOCKS"* state:

Algorithm 1 Round Robin Scheduling Algorithm

```
if $blocks\_scheduled\_cnt < grid\_dimension$ then

  if $blocks\_to\_be\_scheduled >= maximum\_blocks\_occupancy$ then

    for $i = 0, i < SM\_TOT, i++$ do
      $num\_blocks(i) \leftarrow maximum\_number\_of\_blocks\_on\_a\_SM$
    end for

  else

    if remainder of the division between blocks to be scheduled and #SMs is zero then
      for $i = 0, i < SM\_TOT, i++$ do
        $num\_blocks(i) \leftarrow blocks\_to\_be\_scheduled/\#SMs$
      end for
    else

      for $i = 0, i < remainder\_of\_division - 1, i++$ do
        $num\_blocks(i) \leftarrow (blocks\_to\_be\_scheduled/\#SMs) + 1$
      end for

      for $i = remainder\_of\_division, i < SM\_TOT, i++$ do
        $num\_blocks(i) \leftarrow blocks\_to\_be\_scheduled/\#SMs$
      end for

    end if
  end if
end if
```

Inside the *"CALC_BLOCKS"* state, the computation is performed only if the blocks scheduler counter, *"blocks_scheduled_cnt"*, which indicates how many blocks are already scheduled, is lower than the grid of blocks, *"grid_dimesion"*, which stores the number of blocks to schedule in the application. After this first check, another "if statement" follows and two cases can be distinguished:

- the case where the number of blocks to be scheduled is higher or equal than the *maximum blocks occupancy* in the GPU;
- the case where the number of blocks to be scheduled is lower than the *maximum blocks occupancy* in the GPU;

Note that the *maximum blocks occupancy* in the GPU is given by the multiplication between the number of SMs in the design and maximum number of blocks on a SM. Thus, a new signal has been added (*"kernel_blocks_per_core_in_per_SM"*) to store this information.
In the first case, the situation is easier to be managed. The maximum number of blocks for a SM (defined in the configuration file) is assigned to each SM. The remaining blocks will be scheduled when the FSM will come back to the *"CALC_BLOCKS"* state. A counter for the blocks already scheduled, *"blocks_scheduled_cnt"*, and another one for the remaining blocks to be scheduled, *"blocks_remaining"*,

are updated in the following states. For example, consider the Figure 12: 18 blocks must be scheduled on 2 SMs each with 8 as maximum number of blocks. The expression *"Bi"* in the figure indicates the block with the index "i":

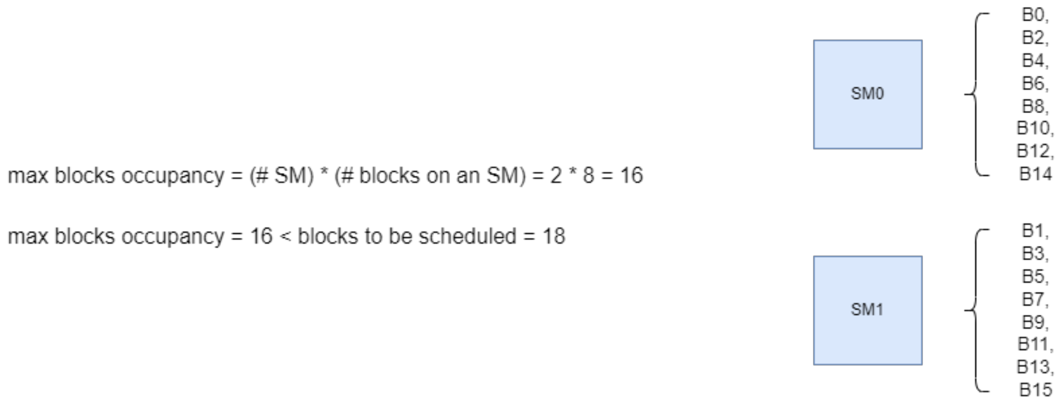
max blocks occupancy = (# SM) * (# blocks on an SM) = 2 * 8 = 16
max blocks occupancy = 16 < blocks to be scheduled = 18
SM0
B0, B2, B4, B6, B8, B10, B12, B14
SM1
B1, B3, B5, B7, B9, B11, B13, B15

Figure 12: Number of blocks to be scheduled higher than maximum blocks occupancy

In this case, first, 16 blocks will be scheduled on the 2 SMs such that the maximum blocks occupancy for each SM is covered. The remaining 2 blocks will be scheduled next (when the execution of the previous 16 blocks is terminated) but, at this point, it falls in the second case.

In the second case (number of blocks to be scheduled lower than the *maximum blocks occupancy*), something more must be done.
The remainder of the division between the blocks to be scheduled and the number of SMs is computed. For this purpose, a new signal, *"blocks_remaining_mod_SM"* is added:

$$blocks\_remaining\_mod\_SM \ <= \ std\_logic\_vector(unsigned(blocks\_remaining) \ mod \ SM\_TOT);$$

where, as told before, *"blocks_remaining"* is the number of blocks not scheduled yet and *"SM_TOT"* is the number of Streaming Multiprocessor in the design.
If the remainder is zero, this means that the blocks can be scheduled equally among all the SMs. At this point, there are no more blocks to be scheduled. Consider the case where 8 blocks must be scheduled on 2 SMs each with 8 as maximum number of blocks, as depicted in Figure 13:

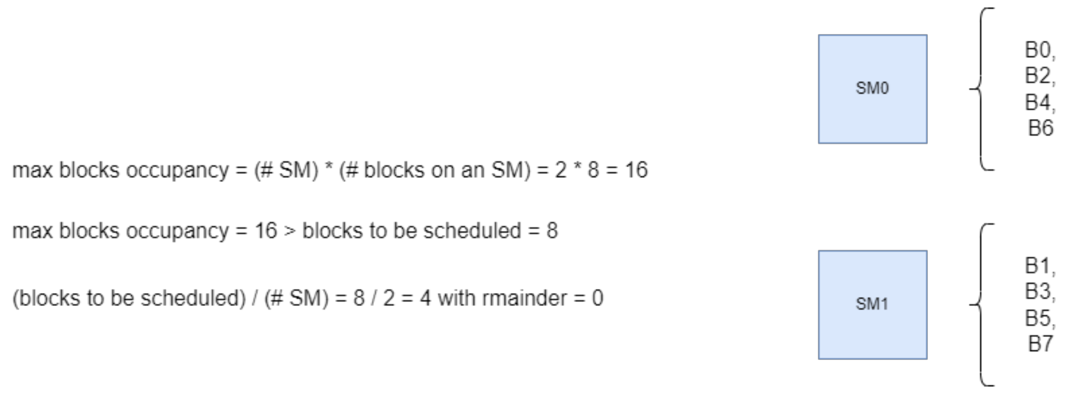
max blocks occupancy = (# SM) * (# blocks on an SM) = 2 * 8 = 16
max blocks occupancy = 16 > blocks to be scheduled = 8
(blocks to be scheduled) / (# SM) = 8 / 2 = 4 with rmainder = 0
SM0
B0, B2, B4, B6
SM1
B1, B3, B5, B7

Figure 13: Number of blocks to be scheduled lower than maximum blocks occupancy and remainder equal to 0

If, instead, the remainder of the previous division is not zero, this means that in this case some SMs will execute more blocks than other SMs. To be more precise, the SMs with an *"id"* lower than the value of the remainder of the previous division will have one additional block to execute with respect to all the other SMs.
For example, consider this case, in Figure 14, where 30 blocks must be scheduled on 4 SMs:

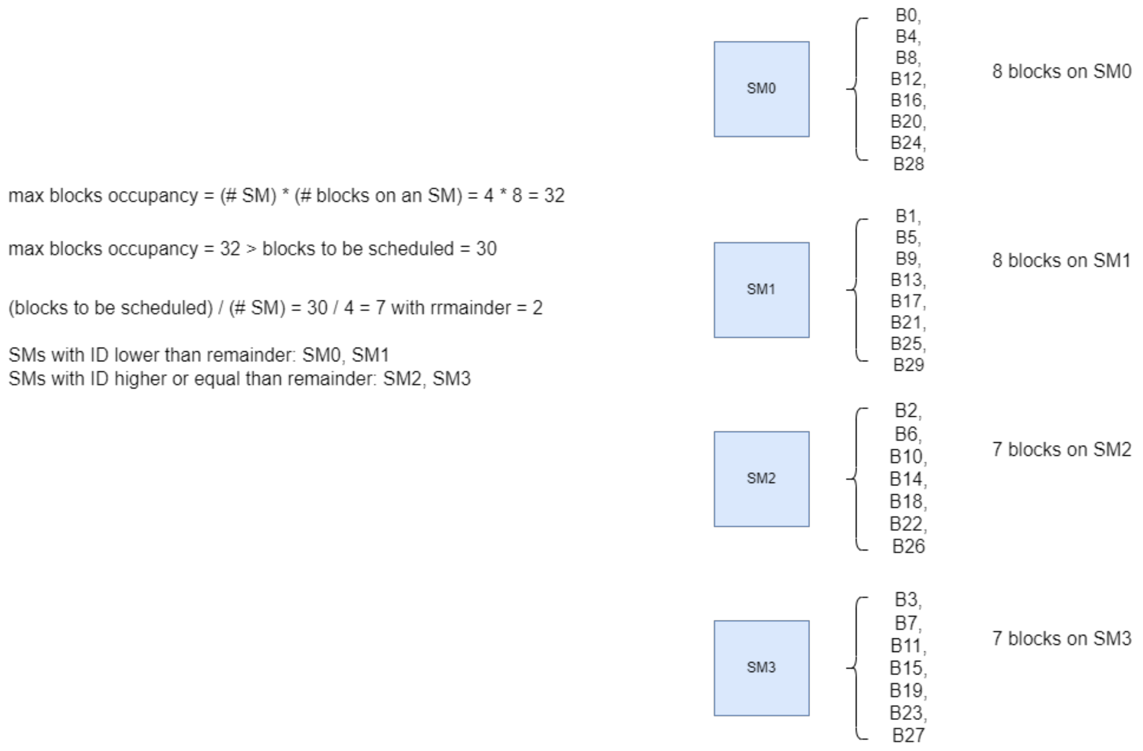
max blocks occupancy = (# SM) * (# blocks on an SM) = 4 * 8 = 32
max blocks occupancy = 32 > blocks to be scheduled = 30
(blocks to be scheduled) / (# SM) = 30 / 4 = 7 with rrmainder = 2
SMs with ID lower than remainder: SM0, SM1
SMs with ID higher or equal than remainder: SM2, SM3
SM0
B0, B4, B8, B12, B16, B20, B24, B28
8 blocks on SM0
SM1
B1, B5, B9, B13, B17, B21, B25, B29
8 blocks on SM1
SM2
B2, B6, B10, B14, B18, B22, B26
7 blocks on SM2
SM3
B3, B7, B11, B15, B19, B23, B27
7 blocks on SM3

Figure 14: Number of blocks to be scheduled lower than maximum blocks occupancy and remainder NOT equal to 0

Also in this case, after this operation, there are no more blocks to be scheduled and so the FSM can go ahead.

The extension of the Block Scheduler for managing a multi-core systems leads to the modification of some signals in the *"SMP controller"*. This last module must use the correct values for the *"id"* of the blocks: these indexes should reflect the changes adopted for the extension of the round robin algorithm in the block scheduler otherwise it could happen that some SMs could execute the same computation. Thus, the initialization and the updating of the indexes *id_i* in SMP controller must be modified correctly. For this reason, in the FSM of the SMP controller, the *"idx_i"* of each block must be initialized to the value *"SM_LOCAL"* which is the variable indicating the id of the Streaming Multiprocessor:

$$idx\_i <= std\_logic\_vector(to\_unsigned(SM\_LOCAL, idx\_i'length));$$

The update of the signal *"idx_i"* is modified as well because of the fact that it should keep into consideration the *"round robin"* algorithm. So it becomes:

$$idx\_i <= std\_logic\_vector(unsigned(idx\_i) + to\_unsigned(SM\_TOTAL, idx\_i'length));$$

At the end, almost 25 % of the descriptions in the scheduler were added or modified.
This step was accompanied by numerous simulations in order to verify the correctness of the model when different configurations of workload, and thus of blocks, where employed in different parallel programs, considering a variable number of SMs in the entire system. In a first instance, the problem of shared resources, such as the memories, was not verified because the main objective was to check the correct distribution of the workload performed by the block scheduler. This method helped a lot in simplifying the verification and the evaluation of the changes in the controller. For this reason, the memories were duplicated and so each SM included a complete memory hierarchy. Obviously, after the simulations, the design was adjusted unifying again the shared resources.

3.2 Memory Management

This step is mainly devoted to make the operations of the SMs coherent and to allow the cores to adequately addressing the same memory to provide correct results in the application. Thus, the SM must access the correct portion of the memories reserved to them avoiding at all costs the contentions due to the fact that the resources are shared.
For this reason, some modules, such as the arbiters and the memories controllers, were modified to allow each SM to access the shared resurce, such as the memory, at the same time. The worst case is when all the SMs want to do an operation (load or store) on the same memory.
This modification is performed in 2 steps mainly:

1. local structural modifications (inside the SM);
2. global modifications (outside the SM);

The first operation was mainly devoted to the modification of the internal structure, such as the control signals in the various stages or the position of the arbiters inside the cores.
The second step, instead, the global modification, required the addition of a certain numbers of new arbiters to handle the global memory request coming from all SMs addressing the global, system and constant memories.

In the FlexGripPlus architecture, the *"Global Memory"* has 2 inputs: the *"input A"* is connected with the Read Stage in the SM, thus is used for reading purposes, while the *"input B"* is connected with the Write Stage and so is used for writing the memory. As said, some structural modifications are needed to deal with a multi-cores system.
As already described in the introduction section, the Read Stage has 3 modules, the "read source operands", for the operands requests to the input A of the *global memory.* Each of these modules is connected with the arbiter present inside the Read Stage. Request, grant and acknowledge are the signals used for the communication among these 3 modules and the arbiter. In the case of single SM, this structure was enough to guarantee correct accesses to the global memory input A. In a multi-core design, this system was not more acceptable. Thus, the idea was to delete the arbiter in the read stage of the SM and to add a single arbiter outside the SM which collects all the requests coming from the "read source operands" of all the SMs configured in the design. This modification is proposed in Figure 15 but a system with only one SM is pictured for sake of simplicity. By the way, if many SMs were present, they will be connected with the arbiter in the same way.

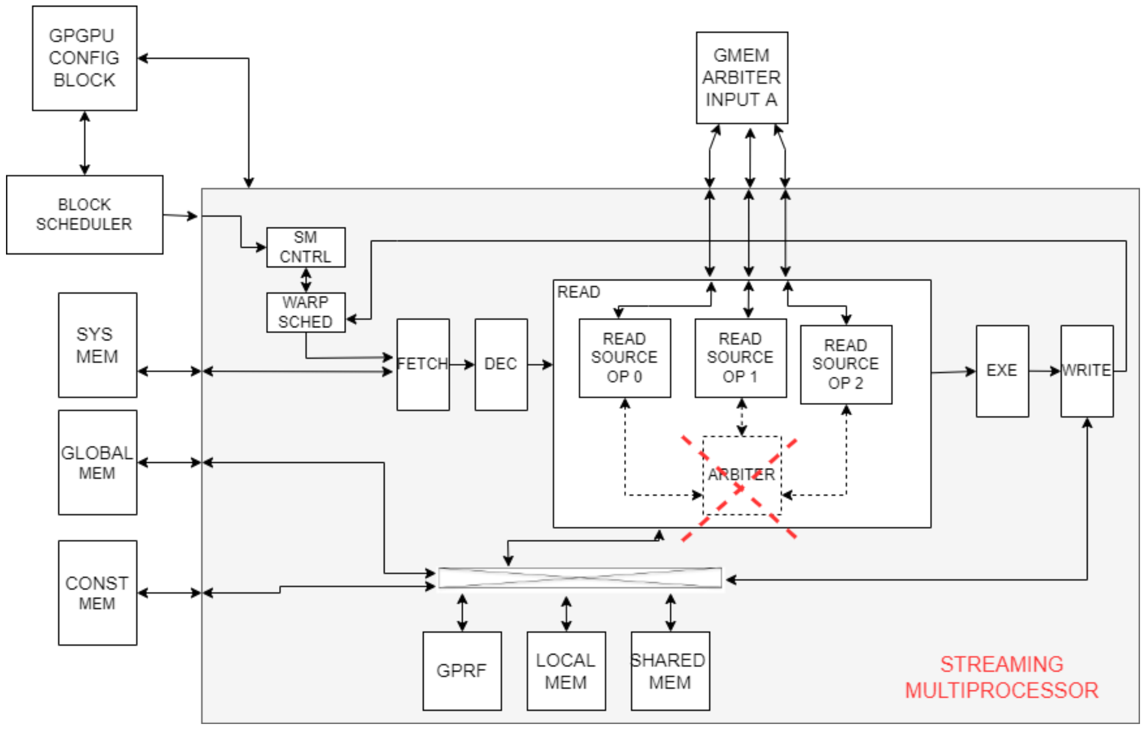

Figure 15: A general scheme of the proposed changes in the architecture for the Arbiter in the Read Stage

The control signals are generated in the Finite State Machine inside each read source operands. These FSMs remain untouched in this modification because they continue to operate in the same way. Obviously, they are now connected with the arbiter outside the SM, so they should be redirect to the correct module. This means that new output and input ports must be added in the SM entity to allow these signals, generated in the inside, to be connected with the outside world.

The input B of the *global memory* is instead connected with the Write Stage of the Streaming Multiprocessor. In the FlexGripPlus, with only one single SM, there is no need of arbiter and, so, neither of control signals. In a design with many cores, instead, a mechanism for the synchronization is needed. First of all, an arbiter is added outside the SMs to collect all the writing requests coming from the cores. Then, the Finite State Machine inside the Write Stage is modified in order to generate the control signals for the requests, the grant and the acknowledges. New states in the FSM has been added to correctly synchronize the operations and to generate in the right instant the request signal: a lot of simulations have been executed to check the correctness of the results written in the global memory. Futhermore, the presence of these new signals leads to the modification of the entity of the Write Stage with the addition of the ports for the communication with the outside arbiter.
At the end, multiplexer and demultiplexer are added to the inputs and outputs of the global memory respectively (as shown in Figure 16). The muxes at the inputs (both A and B) are needed to choose the right data and addresses from the SM which is served by the arbiter. In the same way, the demuxes at the outputs of the global memory are needed to redirect the data to the correct SM. The control

signals for these modules are the grant signals provided by the arbiter.

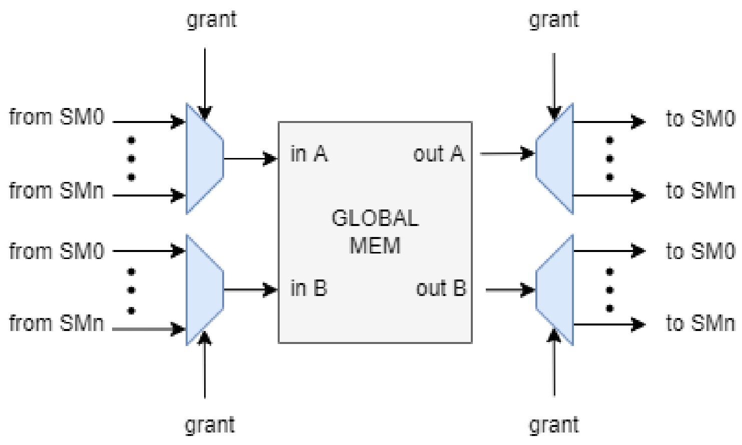

Figure 16: Input and output selection for the Global Memory

The *constant memory* has 2 inputs but only one (input A) is used. The situation is very similar to the one depicted for the input A of the global memory. In the original design, FlexGripPlus, an arbiter is present in the SM which interacts with the same ”read source operands” described before for the global memory. These modules, in addition to generate the control signals for the arbiter of global storage, generate also the signals for arbiter related to the constant memory. Thus the modifications needed are pretty the same: the arbiter is moved outside the SM and now it accepts all the requests coming from all the SMs present in the model. The Finite State Machine inside each read source operands is untouched and the system of muxes and demuxes is added for the input and output of the constant memory. The ports for the control signals are added in the SM entity. This step has been simpler because the management of the situation was very similar to the previous one of the global memory: so a lot of time in simulation has been saved for this purpose.
Another external memory present in the FlexGripPlus is the *System Memory*. It stores the instruction of the kernel and for this reason it is connected with the Fetch Stage. An arbiter is added which has the role to receives all the requests coming from all the SMs. In this case, the Finite State Machine in the Fetch must be modified in order to provide all the control signals for the interaction with the arbiter: new states are added for this purpose. Obviously, ports are added both in the entity of the Fetch stage and in that one of the SM to create the interconnection with the outside world.
At the end, a consideration regarding the *”GPGPU configuration block”* must be done. It is a module used by the FlexGripPlus for the configuration issue at the beginning of the program. It has some connections with the SM. This means that when the number of Streaming Multiprocessor is generic, also some modifications for the gpgpu configuration block are needed. These modifications have the role to avoid eventual contentions among the SMs: thus the same idea exploited for the memories are used for this purpose. The gpgpu configuration block interacts with the Streaming Multiprocessor Controller inside each SM. This means that, as done for other memories, an arbiter is added and the FSM of the SMP controller is changed in order to generate the control signals. Muxes and demuxes are implemented for the input and output respectively of this module and ports are added to the entities.

All these changes in the structure of the FlexGripPlus leads to this new version of the design, represented in Figure 17 (the added units are highlighted in light blue):

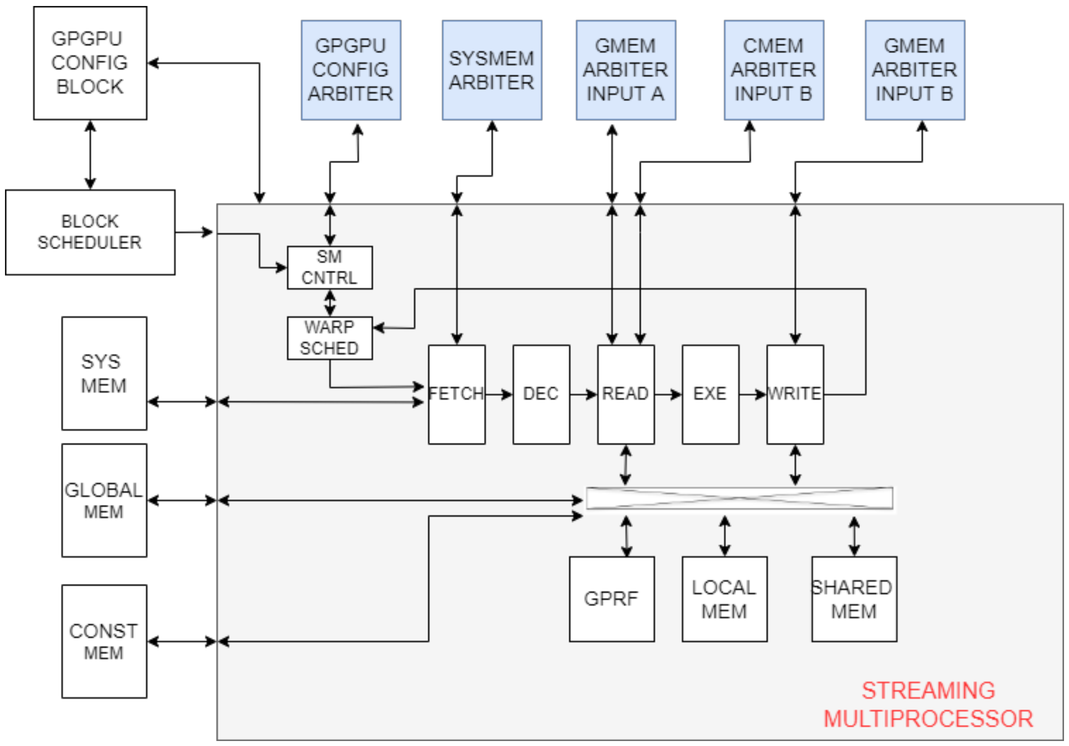

Figure 17: Complete view of the model with the additional arbiters

In Figure 17, only one Streaming Multiprocessor is pictured and muxes are not present for sake of simplicity. The connections between all the stages and the arbiters are shown. If many cores are present, the same interconnections with the arbiters are repeated for all the SMs.
The verification of this step is done performing a lot of simulations to understand if the reading and writing operation were correctly executed in many different cases and on the different kind of storage. For such a purpose, extensive simulations at the RT level were performed to verify the correct operation of the multi-SM GPU with the memory management. At the end, the descriptions of the memory controller modules were modified in 80 % with respect to the previous version.

3.3 Arbiter Handling

The final step (arbiter handling) is focused on adapting the memory controllers to handle simultaneous requests from the SMs and so allowing the synchronization of the operations of the SMs and the end of a parallel program.
For this purpose, the arbiter in the design has been modified to create correct synchronization in the operations. The main limitation of this module was related to the fact, that, when there were more requests at the same time, it served only one device. This is not a proper implementation for the model

where many SMs are present. In fact in this situation, if more SMs request the use of a memory, they all should be served because otherwise some operations will be lost.
To overcome this limitation, a new arbiter (*"arbiter_mod.vhd"*) has been proposed and implemented. It is characterized by this entity presented in Figure 18:

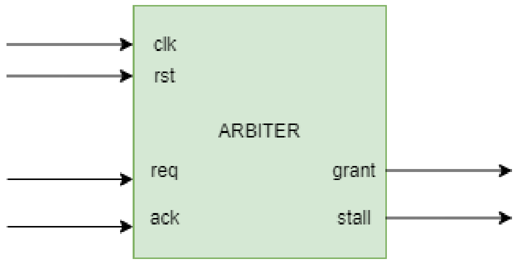

Figure 18: New Arbiter entity

A new signal is used which is the *"stall"*. It is asserted by the arbiter and it is used for synchronization issues. In fact, in this case, if more requests arrive at the same time, the arbiter will serve all of them but one at the time. So it means that when one of the device is served, the other ones should be stalled waiting for the selection by the arbiter. Only when all the requests coming simultaneously are considered, the execution can go ahead.
The implementation is based on a very simple Finite State Machine with only 3 states: *"IDLE"*, *"REQUESTED"* and *"WAITING_ACK"*. In the *IDLE* state, the arbiter waits for the requests generated by the devices. As soon a request arrives, or maybe if more requests arrive simultaneously, the FSM passes in the *REQUESTED* state. At this point one device is served, which means that the grant signal is asserted for it, and, instead, for all the others SMs, the stall signal is asserted. Thus, the FSM goes in the *WAITING_ACK* state: here the arbiter waits for the ack from the SM served. When this acknowledge arrives, this means that the SM has finished its job and so that another SM could be served: so the FSM comes back in the *REQUESTED* state. The FSM will repeat this passages until all the SMs are served. This algorithm is pictured in the Figure 19:

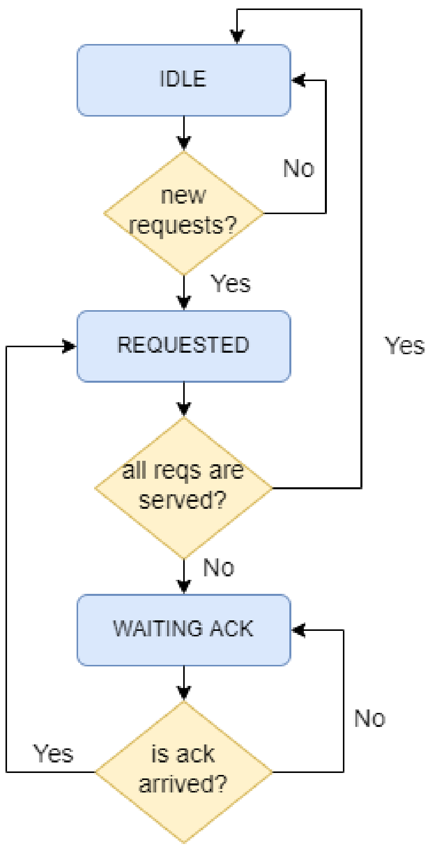

Figure 19: State diagram of the FSM in the arbiter

Notice that the stall signal has been added to the design of the arbiter. This means that all the FSM which interacts with the various arbiters in the design (the ones we described in the previous section) must be modified as well to accept and manage also the stall signal. In the Figure 20, all the connections for the modifications needed for managing the global memory are reported, considering all the control signals (req, grant, ack and stall), all the arbiters, muxes and demuxes. A system with 2 SMs is presented. Obviously, for sake of simplicity, only the interested memory and blocks inside the SMs are represented:

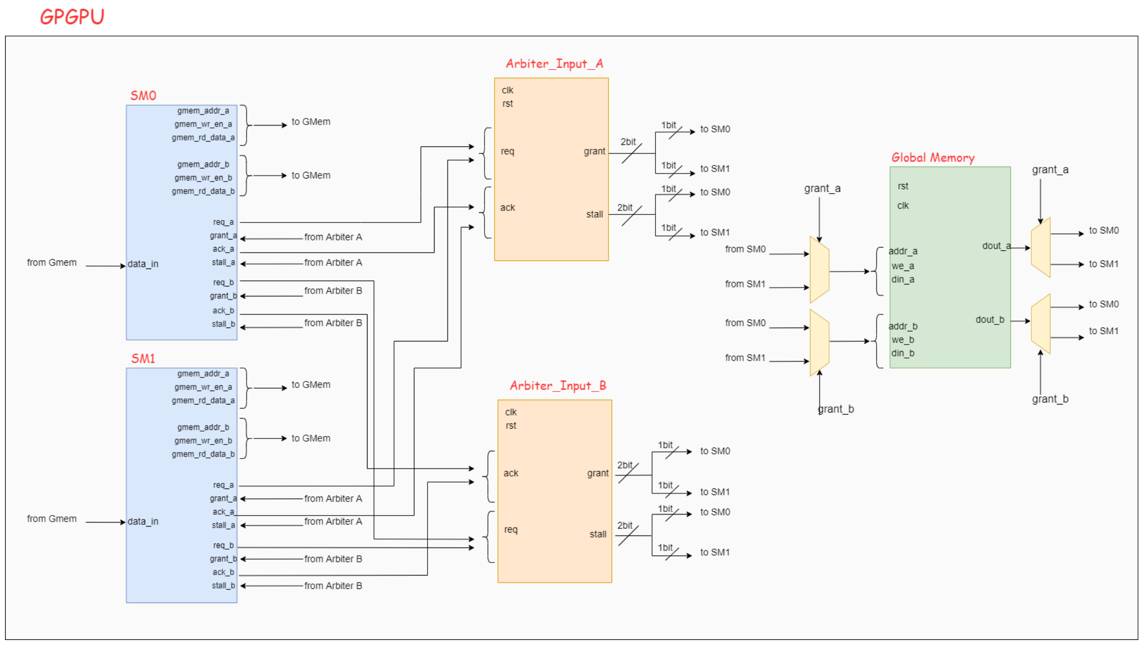

Figure 20: GPGPU with 2 SMs and all the connections for Global Memory Management

In total, about 95 % of the description has been modified for the implementation of the arbiter. This step of the design required different simulations with different configurations to understand if the new design was actually working. In principle the design was tested with 2 Streaming Multiprocessors and only after it was verified with more cores. In any case, the results were compared with the original version with only one SM.

Even if this section is dedicated to the arbiter, it is worth remembering this last modification to the design which is however related to synchronization issues.
Due to the fact that the number of Streaming Multiprocessors is variable, a process is needed to understand when all the cores have terminated their job. This condition is verified when the number of SMs which have terminated is equal to the number of *"active"* SMs. An SM is *"active"* when it has to schedule a number of blocks different from zero. When one SM terminates, a signal is raised (information stored in a vector called *"smp_done_array"*) and a counter *"cnt"* is incremented. Another vector, *"num_blocks"*, is considered: it has an entry for each SM present in the design and this entry is different from zero for all the active SMs. Another counter *"cnt_b"* is used to count the active SMs considering the content of the num_blocks array. When the value of *"cnt"* is equal to the value of *"cnt_b"*, it means that all the active cores have terminated their job. Algorithm 2 shows the pseudo-code of the process:

Algorithm 2 SMs Done

```
$cnt \leftarrow 0$ $cnt\_b \leftarrow 0$
  if $rising\_edge\_clock$ then

    for $i = 0, i < SM\_TOT, i++$ do
      if $smp\_done\_array(i) == 1$ then
        $cnt \leftarrow cnt + 1$
      end if
    end for

    for $i = 0, i < SM\_TOT, i++$ do
      if $num\_blocks(i) /= 0$ then
        $cnt\_b \leftarrow cnt\_b + 1$
      end if
    end for

    if $cnt == cnt\_b$ and $cnt$ $/=$ $0$ then
      $smp\_done \leftarrow 1$
    end if

  end if
```

4 Experimental Results

4.1 Design Validation

4.1.1 Simulation Results

The extension of the FlexGripPlus to obtain a multi-core system for reliability evaluation has been a long process where the modifications were mainly devoted to adapt the system to execute the application on many SMs in parallel, dividing the workload among them. For this purpose, the block scheduler has been modified and the Round Robin algorithm has been adapted. Then, all the memory controllers and the arbiters have been changed to allow the various SMs to access correctly the shared resources (such as the memories). After these structural changes, the arbiter has been handled to synchronize all the operations successfully, avoiding any kind of contention or issue.
For this reason, the validation phase has been crucial in the project (as already repeated in the previous chapters). It was the only way to understand if the process was going in the right direction or not. The tool used was ModelSim: it is a multi-language environment by Mentor Graphics for simulation of models written in a hardware description language (such ah VHDL).
This phase has been the most time consuming one. Each time a new modification was proposed or performed, the simulation followed. An intense set of verifications was useful to check the validity of the new change in the design.
The main idea of the extension was to try to modify the design to first reach a valid model with 2 SMs. Only after that, the final extension for a multi-cores system has been executed. So, first of all, a simplified design has been verified, considering only 2 cores.
Different applications have been simulated to have a complete vision of what was happening inside the processor. The first check was to compare the results stored in the global memory (and printed on an output file) with the results obtained with the original FlexGripPlus design with only one SM. This was the fastest way to verify that the modifications were correct. By the way, after that, a more detailed verification was performed checking the signals inside each interested block on the Waveforms Modelsim tool.
The initial case study has been the *Vector addition* application. Different configurations have been tried with different number of SMs, blocks, threads per block, blocks per core. Notice that, changing the configuration means also changing the number of operations performed, so, as consequence, also the addressing of the global memory must be configured coherently.
In Table 1, different configurations (changing the number of SM, blocks and threads) for the Vector Add application are proposed for testing a multi-cores system.

SMs	blocks	threads	execution time
1	8	64	297,635 ns
1	16	64	593,515 ns
1	32	64	1,185,275 ns
2	8	64	220,555 ns
2	16	64	438,425 ns
2	32	64	875,095 ns
4	8	64	162,735 ns
4	16	64	323,765 ns
4	32	64	644,885 ns
8	8	64	145,055 ns
8	16	64	293,905 ns
8	32	64	592,555 ns
16	8	64	165,295 ns
16	16	64	276,075 ns
16	32	64	563,545 ns

Table 1: Vector Add configurations for design validation

The results shows an improvement in the execution time with the increase of SMs. Obviously, this trend must consider also the application studied and the workload submitted.
Using the values reported in the table 1, the graph reported in Figure 21 is created: it shows the execution time for the Vector Addition application maintaining the same number of blocks (16) and threads per block (64) but increasing the number of SMs (from 1 to 16).

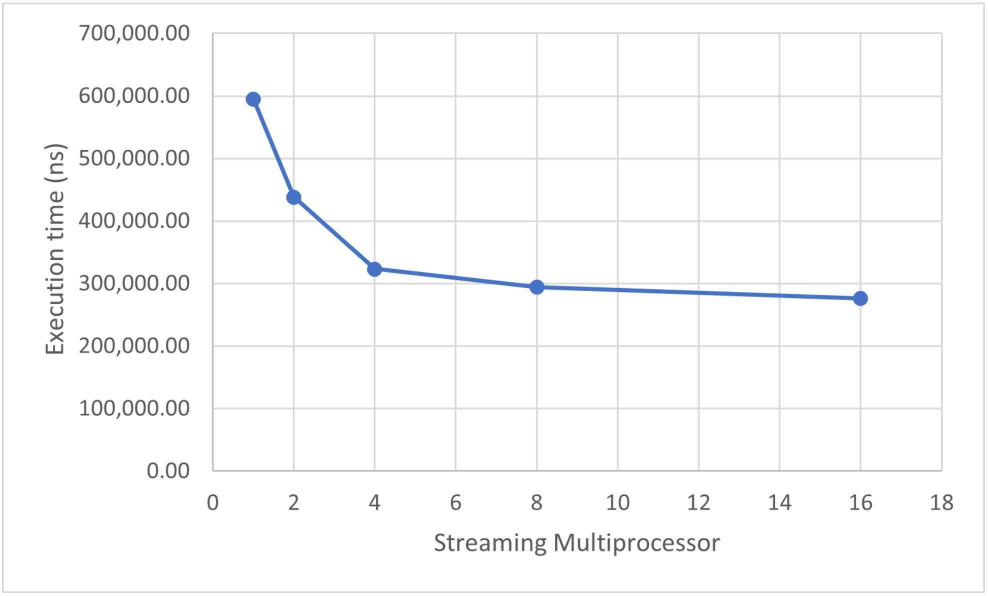

Figure 21: Execution time evaluation for Vector Add with 16blocks and 64 threads per block

Notice that the execution time is calculated considering the "SMP_Processing" phase in the test-bench FSM. It starts after the GPGPU configuration phase and it finishes before the reading memory operations for the results.
Obviously, not only the Vector Add application has been simulated but several other applications. In general results show a proportional relation between the increment in the number of SMs in a GPU and the performance of an application. For all applications, the GPU configuration with 2 SMs allowed a reduction in the execution time (about 26% to 72%) in both block configurations (16 and 32/64) with respect to the configuration with 1 SM.

4.1.2 Design Limitations

The intense set of simulations was useful to find the limitations of this design. The main problem is related to the thread occupancy, which is the maximum number of threads on a single SM. In fact, when the number of SMs starts increasing, the maximum number of threads accepted on an active SM starts decreasing. The Table 2 shows a bunch of simulations with different configurations where 'Y' stands for "consistent result" and 'N' stands for "not consistent results".

1	128	32	128	256	Y
1	64	64	64	512	Y
1	32	128	32	1,024	Y
2	128	32	64	256	Y
2	64	64	32	512	Y
2	32	128	16	1,024	Y
4	128	32	32	256	Y
4	64	64	16	512	Y
4	32	128	8	1,024	N
8	128	32	16	256	Y
8	64	64	8	512	N
8	32	128	4	512	N

Table 2: Set of simulations for Design limitations check

It can be observed that, starting from 4 SMs, the thread occupancy starts decreasing: 1,024 threads on the active SM are not more accepted. For 8 SMs instead, the thread occupancy decreases again and the maximum acceptable value is 512 threads.
When the limit of threads accepted in a certain configuration is exceeded, the threads themselves are no longer correctly positioned in the register file: as a consequence of this situation, the data will no longer be correctly selected and the results, as already highlighted in Table 2, will not be more consistent with those obtained in the original design. This means that, in this new version of the FlexGripPlus architecture, the configuration of the workload for an application strongly depends on the number of SMs established in the GPU.
The reason for the problem could be found in those structures and signals that relate the number of Streaming Multiprocessors and the number of threads present in a given core. In fact, very often during the design phase, some signals with wrong bit sizes were found that could not contain the information correctly due to their limited length. Another cause could be the incorrect organization of threads in the register file; probably, some offsets for the addresses must be considered to allow these memories to properly store all data relating to threads.

4.2 Reliability Analysis

4.2.1 Experimental set-up

The object of reliability analysis is the block scheduler controller which has been extended to manage a multi-cores system. This analysis was not possible before because of the lack of models with an accurate description of the block scheduler in a design with a variable number of SMs. The FlexgripPlus design has been tested in terms of reliability especially in the case of 1 and 2 SMs system and considering only the transient faults (SEUs).
A custom fault injector frameworks is used for this kind of analysis on the block scheduler. It is based on general controller which is in charge of injecting the faults at RT level on a precise and selected module of the model. This framework is hosted by ModelSim tool. To run this reliability analysis, the GPU must execute some application in defined configurations of SMs, blocks and threads. So the workload must be chosen. For this purpose, some applications are selected from the CUDA sample and Rodinia Test suites [24]. Those programs are:

- Vector addition;
- Scalar Product Vector;
- Gray Filter;
- Nearest Neighbour.

The first application computes the addition between 2 vectors storing the results in the global memory. The second application is instead used to perform the scalar product between 2 vectors. Thus, these first 2 programs are one-dimension problems. On the other hand, the Gray Filter is the application used in order to obtain the gray copy of an image starting from its colored version. In this case, the program is on two-dimension differently from the previous ones. At the end, the nearest neighbour application is executed where a proximity search is performed to find a point in a given set which is closest to another point given as input.
Once the applications has been chosen, some free fault simulations are performed considering precise configurations of SM and blocks per SM. This step is useful to determine the starting and finishing time of the execution of the program. The execution phase coincides with the *"SMP_Processing"* state of the main scheduler FSM in the testbench. It includes all the configuration phase of the block scheduler performed at at the beginning but also durign the exchange phase between the active and the waiting blocks.
These timing information are needed for the creation of the fault list. In fact faults will be injected randomly selecting instants belonging to the execution time interval proposed for each application. In the generation of these fault lists, all the signals belonging to the block scheduler structure are considered and the faults will be injected in these locations only. This because the objective of the analysis was the controller only and not the entire design of the FlexGripPlus.
As last step, finally, the fault simulations can be performed. The faults are injected on the signals at the time instant selected by the previous operations. Notice that, only one fault is injected per each fault simulation.
At the end of the fault simulations, the faults are classified in these groups:

- Silent Data Corruption (SDC);

- Detected Unrecoverable Error (DUE);
- Timeout;
- Masked.

SDC means that the fault propagation causes a change in the output results, such that a mismatch is raised between the original fault-free simulation and the faulty simulation.
A *Due* (Crash or Hanging) represents the effect of the hardware fault that once propagated affects the module corrupting the normal execution, thus collapsing the functional operation of the device. In these cases, there are no memory results since the operation is collapsed.
Time out are the faults that once propagated affect the functionality of the device by changing the performance and the execution time of an application but in some cases maintaining correct output results.
At the end, there are the *masked* (or silent) faults: by the effect of the architecture in the device or by effect of the application they are not propagated and do not affect the functionality of the application and the device. These faults are benign and do not compromise the operation of a device.

4.2.2 Experiments performed

The fault injection campaigns for transient faults in the GPU model were performed on a server of 12 Intel Xeon CPUs running at 2.5 GHz and with 256 GB of RAM.
Considering the previous 4 applications presented, for each of them, 4 experiments were performed with different configurations. So, at the end, 16 fault injection campaigns were executed for the reliability analysis of the block scheduler. Others experiments have not been performed because of the fact that this operations are time consuming and a single fault injection can last for several days.
The campaigns were focused on the analysis of the design with 1 or 2 SMs only.
The number of SP cores is variable among the various applications as well the number of blocks. This information are collected in the tables below for each program. By the way, in any case, the applications were configured to force at least once the exchange phase in the blocks scheduling management because, as already explained, this module is more active in this stage.
Notice that, all the applications were configured with 16 and 32 blocks except for the Gray Filter: in this case it was better to consider the case with 16 and 64 blocks because of the two-dimension of the problem.
At the end, almost 111,520 faults were injected during all these campaigns.
The tables 3, 4, 5 and 6 show the configurations used for the 4 different applications in detail.

Vector Add

SMs	blocks	threads	blocks per core	cores	starting time	ending time
1	16	32	8	8	1,725 ns	307,915 ns
1	32	32	8	8	1,725 ns	614,075 ns
2	16	32	8	8	1,725 ns	227,145 ns
2	32	32	8	8	1,725 ns	452,535 ns

Table 3: Vector Add configurations for fault simulations

Scalar Product Vector

SMs	blocks	threads	blocks per core	cores	starting time	ending time
1	16	512	2	8	2,705 ns	14,623,695 ns
1	32	512	2	8	2,705 ns	29,244,655 ns
2	16	512	2	8	2,705 ns	8,471,695 ns
2	32	512	2	8	2,705 ns	16,940,655 ns

Table 4: Scalar Product Vector configurations for fault simulations

Gray Filter

SMs	blocks	threads	blocks per core	cores	starting time	ending time
1	16(4x4)	16x16	4	32	2,105 ns	2,969,455 ns
1	64 (8x8)	8x8	8	32	2,105 ns	2,998,215 ns
2	16 (4x4)	16x16	4	32	2,105 ns	2,350,095 ns
2	64 (8x8)	8x8	8	32	2,105 ns	2,352,295 ns

Table 5: Gray Filter configurations for fault simulations

Nearest Neighbour

SMs	blocks	threads	blocks per core	cores	starting time	ending time
1	16	32	8	32	1,985 ns	244,115 ns
1	32	32	8	32	1,985 ns	486,215 ns
2	16	32	8	32	1,985 ns	182,585 ns
2	32	32	8	32	1,985 ns	363,155 ns

Table 6: Nearest Neighbour configurations for fault simulations

4.2.3 Reliability evaluation

The evaluation starts with the observation of the fault profile of the applications which identifies the fault effects behavior in the scheduler structures. It shows the relation, in time, between the SDCs and DUEs, the injection time and the affected structures. An example is presented in Figure 22 where it is shown the fault profile of the GPU for the Nearest neighbour application with 2 SMs and 32 blocks.

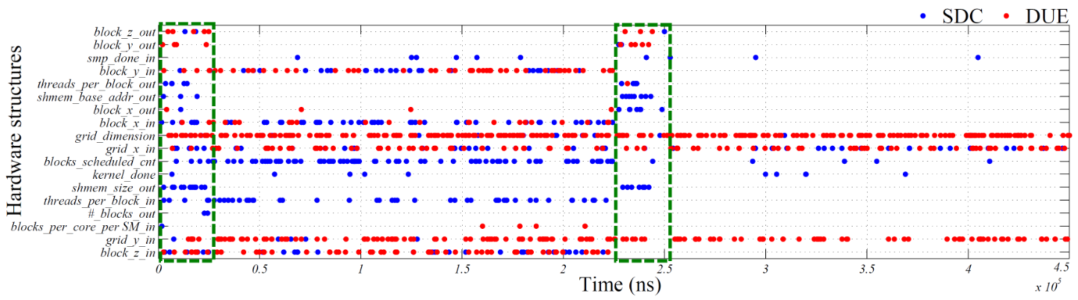

Figure 22: Fault profile for the GPU with 2 SMs and the nn workload with 32 blocks

In the figure 22, it is possible to notice that the faults are mainly propagated in two precise time intervals which are enclosed in the green regions. These stages coincide with the initial configuration phase and the blocks exchange phase, which are the phases where the block scheduler is more active. This trend is observable also in the fault profile of the GPU for other fault simulations executed.
The SDCs are mainly propagated during the initial phase of the execution and until the exchange phase. After this moment, the propagation of the SDCs is highly reduced.
For what concerns the DUEs, instead, Figure 22 shows that not all the structures in the block scheduler are affected by this kind of faults. Some of them are vulnerable to the DUEs along all the execution time, others only during the first phase of the simulation right before the exchange stage and others do not show at all the propagation of this kind of faults.

Another metric used for the reliability analysis is the Architecture Vulnerability Factor (AVF) which is the probability that a fault in that particular structure does not result in an error. It is obtained dividing the number of faults propagated by the number of total faults injected in the fault simulation. It is computed for the SDCs, the DUEs and the Timeout faults. In Figure 23, the AVF results for the fault injection campaign are shown.

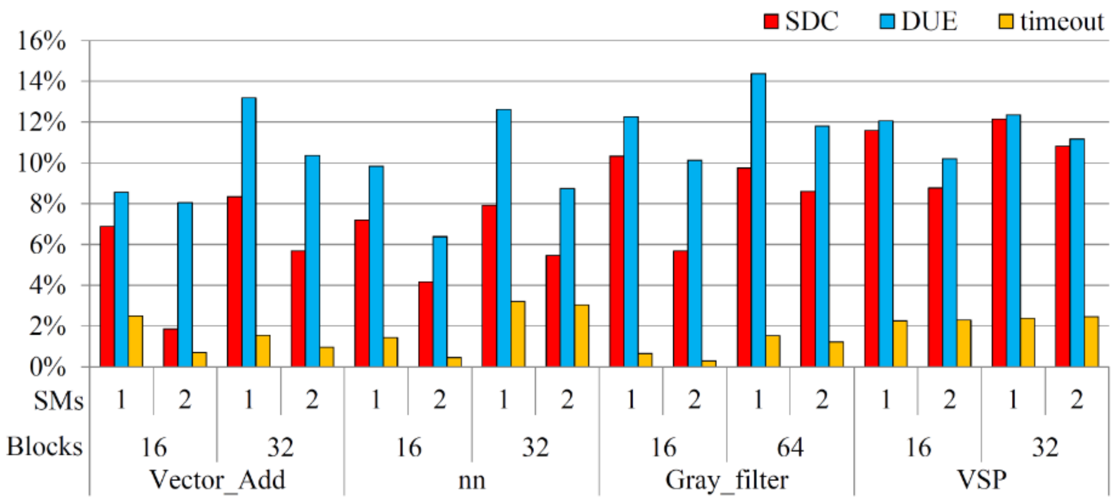

Figure 23: AVF results for several workloads affected by faults in the block scheduler

The results shows that a system with 2 SMs is more reliable than a system with 1 SM only. In fact the values for the AVF decrease in the transition from 1 core to 2 cores. The only case where this trend is not respected is for the timeout faults in the vector product application.
Furthermore, Figure 23 shows that a configuration with more blocks seems to be less reliable than a configuration with less blocks.

At the end, the Mean Time Between Failures (MTBF) is considered: it is a very common indicator and parameter of reliability in maintenance management of devices. It describes the average time between failures, or the average time expected between the start of a failure and the start of the next failure. It is used to take into account the execution time (expressed in clock cycles) of the program in the reliability analysis consideration. The higher the MTBF, the longer a system is likely to work before failing. Measuring MTBF is one way to get more information about a failure and mitigate its impact. Conducting an MTBF analysis helps to reduce downtime while saving money and working faster. This metric is computed for the SDCs only. The expression is given by:

$$MTBF = \frac{1}{cross\ section\ \cdot\ f}$$

where f is the flux of particles affecting the module per time unit and it is given by:

$$f = \frac{1}{total\ simulation\ time}$$

while the *cross section* is given by the following expression:

$$cross\ section = \frac{detected\ faults}{total\ injected\ faults}$$

In the Figure 24, the results for all the workloads related to this metric are shown.

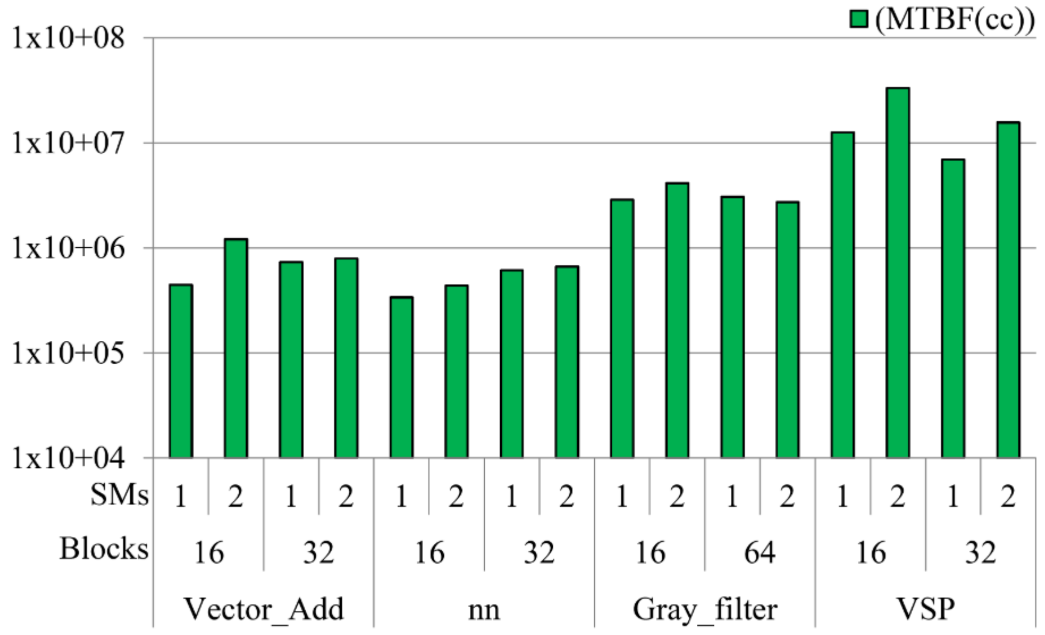

Figure 24: MTBF results (in base-10 logarithmic scale) for the workloads

It is possible to repeat the same observations done with the AVF metric. In fact, it seems that, with the same number of blocks, the configurations with 2 SMs are more reliable than the configurations with 1 SM (MTBF is higher in the first case). By the way this trend is not respected in the gray Filter application with 64 blocks because the configuration with 1 SM seems to be more reliable. This results depend on the different possible configurations in the different applications.

5 Conclusions and future works

This thesis work is focused on the reliability analysis of the main controller of a GPU device, the Block Scheduler, to determine the effects of the transient faults on this module. To do that, an open-source low-level micro-architectural model based on the NVIDA G80 architecture has been extended to allow the execution of the applications on a multi-core system.
The design stage has started by the extension of the Block Scheduler to make it suitable for an architecture with a variable number of Streaming Multiprocessors. Thus, the internal implementation of such a module has been modified: the Round Robin algorithm for the scheduling of the blocks has been adapted to deal with many cores.
After that, the structure of the design has been changed to allow the SMs to access the shared resources (memories and GPGPU configuration block) without creating any contention issue. For this purpose, arbiters have been added in the architecture to manage all the requests coming from the cores and the controllers in the SMs have been modified with new control signals to allow correct communication with the arbiters.
At the end, the internal implementation of the arbiters have been modified and extended to correctly synchronize the SMs requests of the shared resources.
Each modification was followed by an intense set of simulations performed on ModelSim tool to check the correctness of the design. The model has been tested with different parallel applications on different configurations of SMs and blocks. The main limitation of the architecture is related to the fact that when the number of SMs increases, the maximum thread occupancy in the SM decreases. If the maximum thread occupancy in the different situations is not respected, the results of the application will be wrong. Thus, the configurations of workload strongly depends on the number of SMs present in the system.
Only after the design stage was over, the reliability analysis has started. In particular, the evaluation was performed on the block scheduler considering only transient faults and for four different applications: Vector Addition, Gray Filter, Scalar Product Vector and Nearest Neighbour. For each of these programs, a fault injection campaign was performed with 4 different configurations: 1 SM with 16 blocks, 1 SM with 32 blocks, 2 SMs with 16 blocks and 2 SMs with 32 blocks. Only for the Gray filter, because of the 2-dimension of the problem, instead of 32 blocks, 64 blocks were used.
Results shows that the configurations with 2 SMs seem to be more reliable than the configurations with only 1 SM. Furthermore, the vulnerability of the block scheduler strongly depends on the block configuration chosen for the application. These considerations were based on the evaluation of two different metrics: the Architecture Vulnerability Factor (AVF) and the Mean Time Between Failures (MTBF). The fault profile of the different simulations has been studied and it shows that the design is mainly affected by the faults during the initial configuration phase of the block scheduler and during the exchange phase.
Future works may move towards two different directions: overcoming the design limitation of the actual FlexGripPlus and extending the reliability analysis. The first case will be focused on the correction of the actual model to avoid the problem related to the maximum thread occupancy decreasing with the increasing of the number of SMs. On the other hand, the second case will be dedicated to the evaluation of different fault models in the scheduling controllers and to extend the analysis to a larger number of applications with different workload configurations.

References

[1] W. Shi, M. B. Alawieh, X. Li, and H. Yu. *"Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey". Integration*, 2017.

[2] D. Hallmans, K. Sandstrom, M. Lindgren, and T. Nolte. *"Gpgpu for industrial control systems". 2013 IEE 18th Conference on Emerging Technologies Factory Automation (EFTA)*, 2013.

[3] A. Alic, S. Visan, and R. Potolea. *"Towards fast bioinformatics algorithms: Benchmarking the gpgpu". 2011 10th International Symposium on Parallel and Distribuited Computing*, 2011.

[4] M. C. Diaz, F. A. Gonzalez, and R. Ramos-Pollan. *"Accelerating common machine learning algorithms through gpgpu symbolic computing". 2015 10th Computing Colombian Conference (10CCC)*, 2015.

[5] V. Campany, S. Silve, A. Espinosa, J. C. Moure, D. Vazquez, and A. M. Lopez. *"GPU based pedestrian detection for autonomous driving". Procedia Computer Science 80*, 2016.

[6] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, and Y. Kitsukawa. *"Autoware on board: enabling autonomous vehicles with embedded systems". ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, 2018.

[7] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith. *"Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems". 30th Euromicro Conference on Real-Time systems (ECRTS 2018)*, 2018.

[8] J. E. R. Condia, B. Du, M. Sonza Reorda, and L. Sterpone. *"Flexgripplus: An improved gpgpu model to support reliability analysis". Microelectronics Reliability*, 2020.

[9] J. Sanders and E. Kandrot. *"CUDA by Example, An Introduction to General-Purpose GPU Programming"*. 2010.

[10] M. J. Flynn. *"Some computer organizations and their effectiveness". IEEE transactions on computers*, 1972.

[11] S. H. Roosta. *"Parallel processing and parallel programming". Parallel Processing and Parallel Algorithms*, 2000.

[12] M. Merchant. *"Testing and validation of a prototype GPGPU design for FPGAs". M.S. thesis, University of Massachussestts*, 2014.

[13] J. Owens and D. Luebke. *"The Gpu programming model". Udacity online course.*

[14] K. Andryc, M. Merchant, and R. Tessier. *"FlexGrip: A soft GPGPU for FPGAs". 2013 International Conference on Field-Programmable Technology (FPT)*, 2013.

[15] J. E. R. Condia. *"New Techniques for On-line Testing and Fault Mitigation in GPU". Doctoral Program in Computer and Control Engineering (XIII Ciclo)*, 2021.

[16] S. Costa. *"Study and development of a VHDL infrastructure for signals probing of GPGPU architectures". M.S. thesis, Politecnico di Torino*, 2020.

[17] G. Roascio. *"Analysis and extension of an open-source VHDL model of a General-Purpose GPU". M.S. thesis, Politecnico di Torino*, 2018.

[18] Mohammadmahdi Mohammadi. *"Evaluating the impact of counters for the in-field test of GPUs". M.S. thesis, Politecnico di Torino*, 2021.

[19] C. A. Chami. *"VHDL Arbiter". FPGA Site*, 2018.

[20] J. C. C. Nelson and M. K. Refai. *"Design of a hardware arbiter for multi-microprocessor systems". Microprocessors and Microsystems: Embedded Hardware Design (MICPRO)*, 1984.

[21] S. Di Carlo, G. Gambardella, I. Martella, P. Prinetto, D. Rolfo, and P. Trotta. *"An improved fault mitigation strategy for CUDA Fermi GPUs". Dependable GPU Computing workshop*, 2014.

[22] K. Cho and H. Bahn. *"Performance Analysis of the Thread Block Schedulers in GPGPU and its applications". Applied Science*, 2020.

[23] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. *"NVIDIA Tesla: A Unified Graphics and Computing Architecture". IEEE Micro*, 2008.

[24] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. *" Rodinia: A benchmark suite for heterogeneous computing". 2009 IEEE international symposium on workload characterization (IISWC)*, 2009.