POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING (DAUIN)

Master Degree in Computer Engineering

Master Degree Thesis

Analysis and extension of an open-source VHDL model of a General-Purpose GPU

Author: Gianluca Roascio

Supervisiors: Matteo Sonza Reorda, Luca Sterpone, Boyang Du

October, 2018

A Vittorio, Diamantina, Attilio ed Anna, i miei quattro angeli custodi, tre in Cielo ed uno in Terra.

Contents

Chapter 1

Introduction

In the last years, high-performance computing requirements led to unite the advantages brought by parallel computing typical of graphic processors with the flexibility of the general-purpose programming. This is how the so-called GPGPUs (General-Purpose Graphic Processing Units) have started to be developed and diffused on the market. These units present themselves as parallel processors with the natural tendency to treat large blocks of data. This large amount of computation is in fact spread among the different *multiprocessors*, single units containing multiple *cores* inside. All cores commonly execute the same code, but just with different portions of inputs. This architecture paradigm is called SIMD (Single Instruction Multiple Data).
GPGPUs even extend the concept of SIMD combining it with low-level multithreading. This execution model is called SIMT (Single Instruction Multiple Thread): instead of just having a single program flow which contains instructions addressed to packed data, here every instruction is executed by different threads which have real hardware consistency. This means that each of them is not created at software level as in the CPU multithreading, but has a reserved portion of resources (ALU, register file) over which it processes its assigned data. For these reasons, in a GPGPU most of the silicon area is devoted to data processing units, with only a relatively little portion for caching and control. In these conditions, throughput reaches levels unachievable for classical microprocessors, even if belonging to last generations.
GPGPUs are difficultly used as stand-alone units, while they are more usually placed alongside main CPUs (which play the role of masters of the system) which feed their device storage with both instructions and data for those routines which may have strict requirements in terms of acceleration and parallelization. These routines take the name of *kernels*.
It is undoubtedly the Nvidia's CUDA platform the master in the CPU-GPU cooperation framework market. CUDA (Compute Unified Device Architecture) is a set of APIs which extend the common C++ libraries to allow programmers not to care about underlying graphical concepts. A common CUDA program thus contains a control part executed by the CPU and a computational kernel executed by the GPU.
However, due to their rigid microarchitectural features, GPGPUs are not available as support unit for most of the computing systems. On the contrary, FPGAs (Field Programmable Gate Arrays) are can be seen as soft (reconfigurable) hardware accelerators to be placed next to CPUs, because they are perfectly adaptable to the designer requirements. However, when using FPGAs a certain degree of expertise in HDL (Hardware Description Language) coding is expected, and besides, reconfiguration of the logic may take unaffordable times.
In order to put together the pros of the two platforms alleviating their drawbacks, the University of Massachusetts has developed a model for a *soft* integer GPGPU optimized for FPGA implementation which is called **FlexGrip (FLEXible GRaphIc Processor)**. FlexGrip is based on the G80 architecture by Nvidia, the first dedicate general-purpose architecture by the company, and allows the direct execution of CUDA binary code with Compute Capability 1.0. FlexGrip supports 28 base instructions declinable in different formats, among which arithmetic and logic operations, movement between the different levels of memories, branches and thread synchronism setting and predicated instructions. The internal parallelism can be set at different levels, such as the number of parallel internal cores within

the multiprocessor.
Beyond the benefits of using it as an accelerator for FPGA, FlexGrip also has the trivial but not obvious merit of offering the academic community an open-source model of a graphic processor, due to the confidentiality issues adopted by GPU manufacturer about the implementation details of their devices. The accelerated development that some sectors of the industry have known in the last years from the automation point of view (think for example of the automotive industry) necessarily pushes companies to request that these components have a sufficiently high degree of *reliability*. The challenge is gathered by the research centers and the academic teams involved in IT reliability, such as ours, which are trying (among the other activities) to offer innovative techniques for building efficient SBST (Software-Based Self Test) programs for such devices. In fact, many of the techniques used routinely for the self-test of normal processors cannot be reused here, due to the presence of specific modules that are to be precisely studied at the gate level to identify all the possible faults. This is clearly not possible without the netlist of a working GPU.
Then, this thesis work has been focused on the validation, debug and extension of the VHDL model of FlexGrip through an instruction-level analysis. The behavior of all the nominally supported functionalities has been analyzed by compiling a high number of random CUDA kernels to consider as many formats as possible of each of the instructions, due to the absence of any reference manual for the effective ISA by Nvidia. Formats which were not fully working or implemented have been corrected and completed, as well as collected in a tool which is able to write down the corresponding mnemonic and binary in the .SASS input file (the format of the assembly language for the Nvidia family).
The present document is organized as follows: Chapter 2 provides a general background on FlexGrip in terms of architecture and functioning. Chapter 3 describes the activities performed in these months to bring the model to the current status. Chapter 4 is the final chapter, that includes some points of evaluation of the performance of the model and ideas for future development.

Chapter 2

Background on FlexGrip

2.1 General Architecture and Functioning

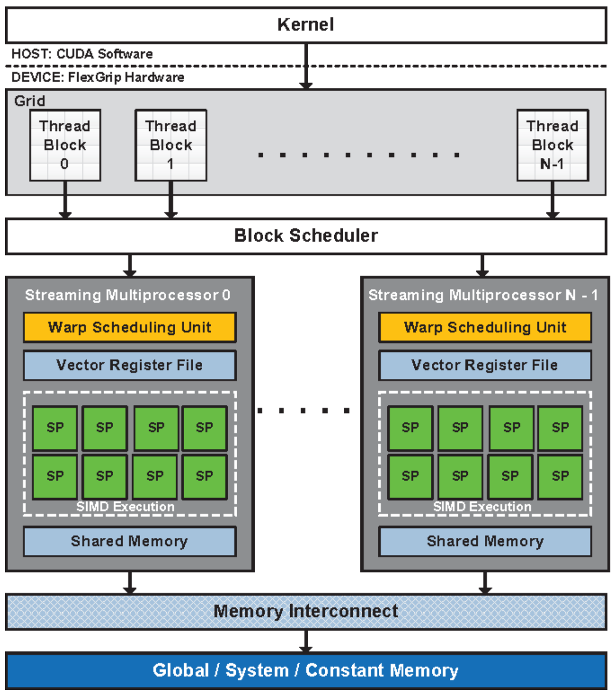


Figure 2.1: Overview of a GPGPU architecture.

The Figure 2.1 offers a global overview of FlexGrip architecture, and in general of a GPGPU. It basically has a multicore architecture, consisting in an array of *streaming multiprocessors*, each one containing a certain number of parallel *scalar processors* which enable the device to execute more threads in parallel. Multiprocessors are called *streaming* because each of them is composed by processing elements that perform the same operation on multiple data simultaneously. Every thread is mapped onto a single scalar processor, which provides to it dedicate arithmetic and logic resources to perform its task. Such cores generally use 32-bit operands, but can also work with halfwords (16 bits). Each thread has then reserved for it a portion of the *vector register file*, a significant amount of immediate memory to be used as dedicate general-purpose registers, where to store immediate results. All threads can access in parallel to this storage component, but no one of them can access to a portion assigned to another thread. The communication between threads belonging to a block is achieved through the *shared memory*. At the bottom of the memory hierarchy there are the interfacing memories such as

the *global memory*, which is visible by all SMs and usually stores inputs and outputs of the kernel for the communication with the host, the *constant memory* which is a read-only space where the kernel constants are written by the host, and the *system memory* which contains the instructions of the kernel. Obviously, unlike these types of memory, the overall amount of memory internal to an SM for implementing the vector register file and the shared memory is divided into the number of threads and blocks, so it sets an upper bound to the reachable internal parallelism.
At the hardware level, the kernel is launched by the host which decides the number of blocks present in the overall *grid* and the number of threads composing each block. A block contains a certain number of threads that can cooperate together, for this reason it is also called CTA (Cooperative Thread Array). The grid is seen as a two-dimensioned set of blocks, and the blocks are seen as three-dimensioned sets of threads. The programmer can also decide the length of all these dimensions to organize the parallel configuration. Once composed, the grid is passed to the *block scheduler* which dispatches the blocks among the available SMs. At its internal, the multiprocessors further split the blocks organizing them into subset of threads called *warps*. A warp is a smaller set of simultaneous operations that may be performed conditionally, generally 32-thread wide. The number of scalar processors within the SM normally fits the warp dimension, but if this is not happening, warps are divided into *lanes* of threads executed one after the other in parallel on the available SPs.
When the SM fetches a new instructions, through the employ of the *warp scheduler* assignes it to a warp which is marked as READY. Warps can also be ACTIVE if they are already executing an instruction, WAITING for other warps to reach a common synchronization point (the mechanism will be shown in next sections) and possibly FINISHED. Every thread composing the warp addresses a different portion of data, and it is free to take data-dependent branches or it can be stopped from execution by data-dependent conditional instructions. Summing up, a warp is characterized by its state, its own program counter and a *thread mask* which indicates how many threads are executing the instruction corresponding to that PC. Depending on the value of each bit in the mask, a scalar processor is enabled or inhibited from execution of the thread assigned to it.
Whenever a divergency is created inside a warp, instructions pointed to each subset of threads have no other solution but to be executed serially. This results in a non-full utilization of the resources which obviously penalizes the performance. The worst case scenario is when each thread of the warp takes its own branch, and an $O(n)$ performance penalty is caused.
In order to summarize the software-hardware interaction which makes possible the execution of the kernel on the GPU:

- The kernel passed by the host is mapped onto a grid of CTAs (blocks)
- The block scheduler assignes each block to a streaming multiprocessor
- The multiprocessor further divides the assigned blocks into warps
- At each new instruction, the warp scheduler selects a ready warp for its execution
- Within the warp, each thread is assigned to a single scalar processor

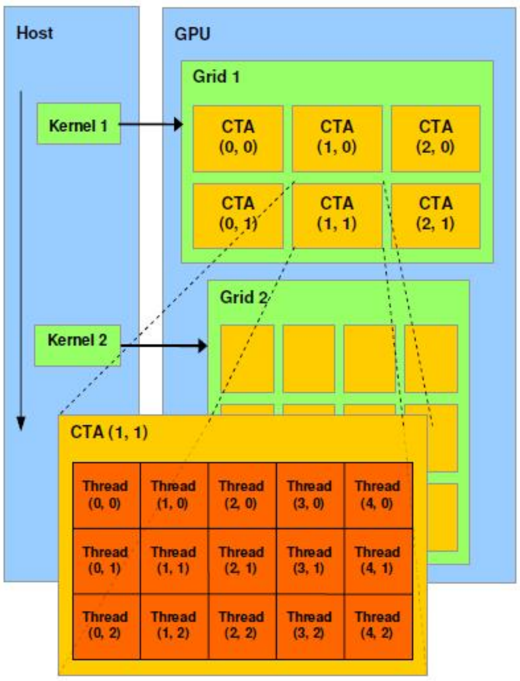


Figure 2.2: Host-GPU interaction.

2.2 Streaming Multiprocessor Architecture

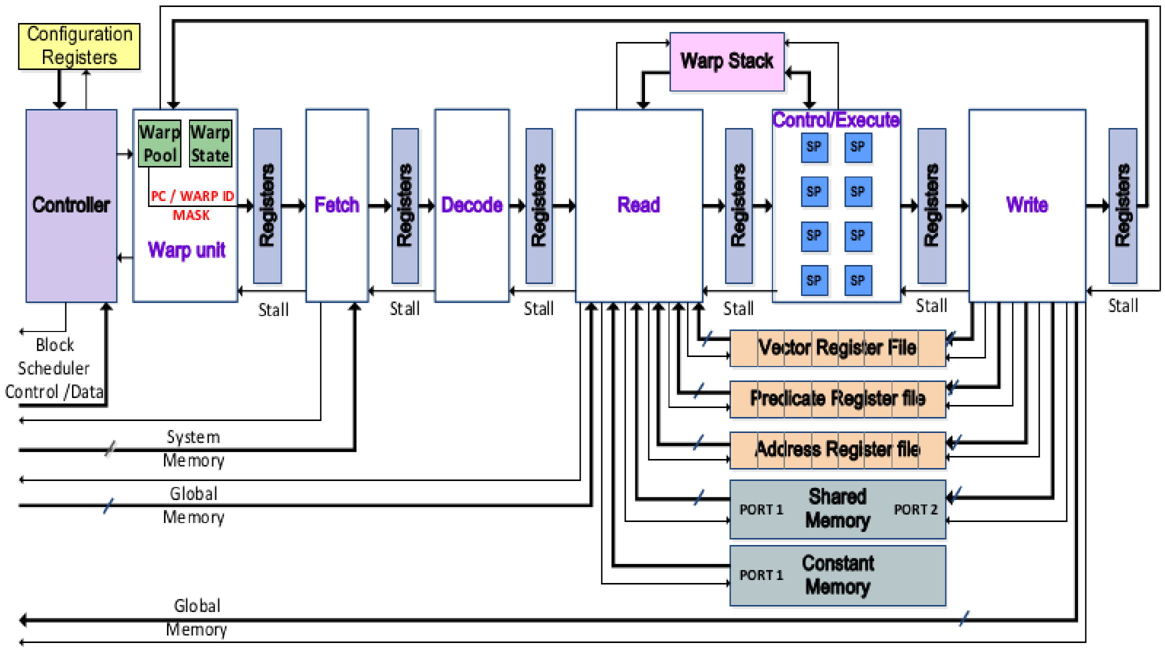
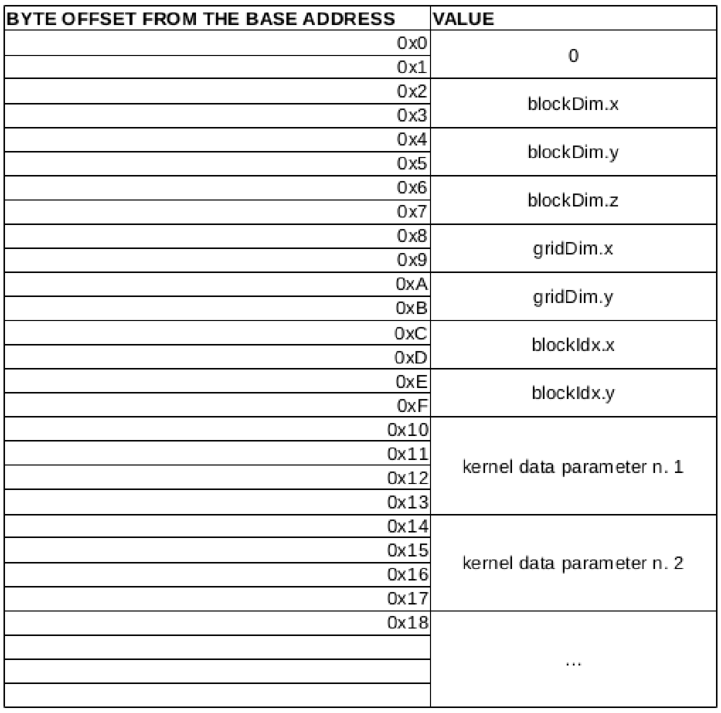


Figure 2.3: SM internal architecture.

Let's move now the focus to the internal pipeline which characterizes each streaming multiprocessor. It is designed as a five-stage pipeline (*Fetch, Decode, Execute, Read* and *Write*) which recalls the MIPS architecture. Unlike MIPS, many instructions can retrieve at least one of their operands directly from the shared or the constant memory (*register-memory architecture*). This is noticeable in Figure 2.3: the Read stage preceeds the Execute one, and has connections with all types of data storage.

2.2.1 Control Units

At configuration time, when the SM is not yet enabled, the external driver loads in all the *kernel parameters* necessary for its execution, such as the dimensions of the grid and the blocks, the number of blocks per multiprocessor, the number of used general-purpose registers and the size of the shared memory per block. It also delivers the actual parameters of the kernel function (for the CUDA programming model, as it will be explained later on, a kernel is still a procedure called by host, so it owns a set of data parameters) and sets its number.
The supervision of the whole SM is entrused to the *streaming multiprocessor controller*, which is the module that receives the "go" command from the block scheduler. When the controller wakes up, it finds all the configuration parameters mentioned before both on the signals from the block scheduler and in the *configuration registers*, which are a portion of storage external to the SM. 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*. Such header contains the length of the 3 dimensions of the block, the 2 dimensions of the grid either, the x and y indexes of the block within the grid, and the list of all the data parameters of the kernel, one after the other. While the block parameters are organized in words of 16 bits, parameters are on 32 bits.



BYTE OFFSET FROM THE BASE ADDRESS	VALUE
0x0	0
0x1	
0x2	blockDim.x
0x3	
0x4	blockDim.y
0x5	
0x6	blockDim.z
0x7	
0x8	gridDim.x
0x9	
0xA	gridDim.y
0xB	
0xC	blockIdx.x
0xD	
0xE	blockIdx.y
0xF	
0x10	kernel data parameter n. 1
0x11	
0x12	
0x13	
0x14	kernel data parameter n. 2
0x15	
0x16	
0x17	
0x18	...

Figure 2.4: The block header in the shared memory.

Once initialized each block in the shared memory, the controller must initialize the threads in the vector register files. For each scalar processor present in the SMP, there is also a dedicate dual-port register file of each of the 3 types (*general-purpose, address* and *predicate*, their usage will be explained later on). Because of the presence of multiple blocks and warps, threads are much more numerous than single register files, so the controller computes a *base address* for each thread. At this location, it writes the x, y and z coordinates within the block as follows.

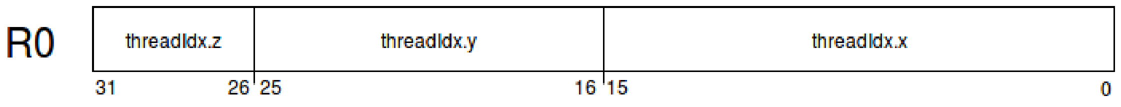


Figure 2.5: Register 0 filling with thread IDs.

The control is now passed to the *warp unit*, which receives from the controller the number of blocks assigned to the SMP and so it determines, in base of the warp size, how many warps are to be generated. For each warp, a *warp pool lane* of 128 bits is generated and stored in a dedicate portion of memory inside the unit, addressed with the warp ID and composed as follows.

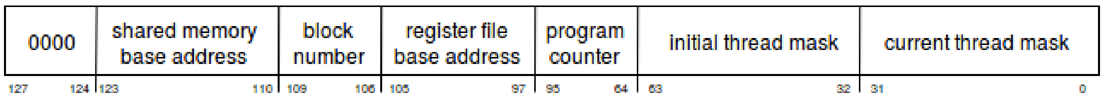


Figure 2.6: The warp pool lane fields.

Also, the state of each warp is stored in a second portion of memory. A third storage unit, the *fence registers*, is used to mark what are the warps waiting at a *block barrier synchronization point*: this point is set by a dedicate instruction and makes all warps belonging to a block synchronize at a certain instruction pointer. All warps executing such instruction must be set as waiting and unlocked only when all warps have executed it.

The warp unit schedules warps in round-robin fashion, reading the pool and the state. The pipeline is enabled when a READY warp is scheduled. Then, its state is set to ACTIVE and its parameters are passed to the Fetch stage.

2.2.2 Fetch and Decode

The instruction bus towards the system memory is 32-bit wide, so depending on the length of the instruction (32-bit if half or 64-bit if full), 1 or 2 cycles are taken by the Fetch stage to complete the read-in. The next program counter is incremented by 4 or 8 and the instruction is passed to the Decode stage.
The Decode is a pretty complex module which, depending on the opcode of the instruction, individuates fields for source memory type, address or register number, size of the operands, conditional fields, instruction pointer values (in case of branches or synchronization point setting) and others. Fields' position and dimension are not strictly fixed and depend on the instruction, as it will be explained in the next Chapter, when the ISA will be presented.

2.2.3 Read of the operands

The Read stage has 3 internal modules which work simultaneously and are able to retreive in parallel the operands from the register files and serially from the other memories. Let us see in details how the register files are organized and their tasks.
The vector register files are the first type of fast memory dedicate to single cores, and are used to implement general-purpose registers, as already said. For each SP, a bank of **512 32-bit registers times the number of warp lanes** is allocated. An example is reported in the image below: in this configuration, 8 cores are present in the SMP, so 4 warp lanes (rows) are needed. The numbering from 0 to 31 indicates the index of the thread within the warp.

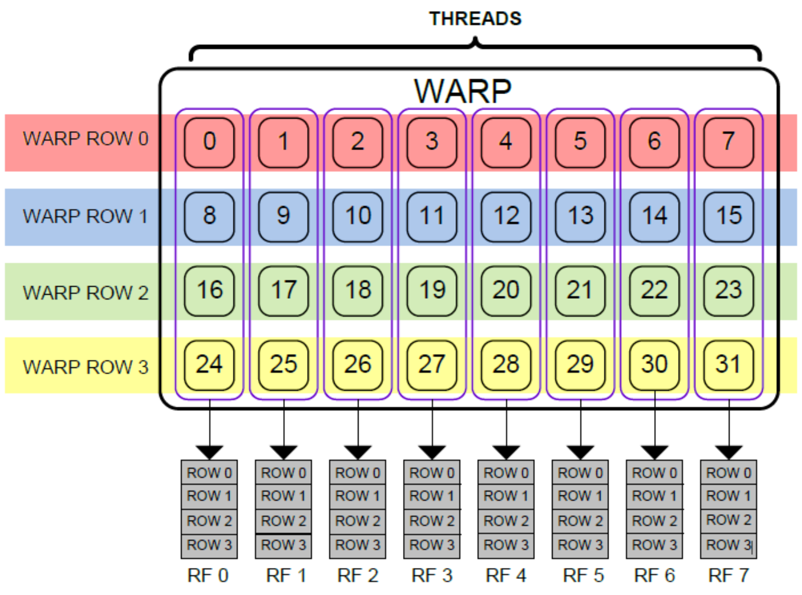


Figure 2.7: Register file configuration with 8 cores.

If 16 cores were there, RF from 0 to 15 would be present, just row 0 and row 1 would appear, and each register file would be composed of 1024 instead of 2048 locations.
Each RF is dual-ported, with the port A contended by the SMP controller and the Read stage, and the port B left to the Write stage. Being 2048 the maximum depth of each RF, its address busses are 11-bit wide, organized as follows.

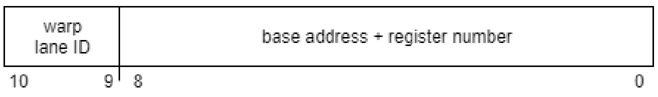


Figure 2.8: General-purpose register address composition.

The number of registers per thread is decided by the driver at configuration time, anyway it cannot go beyond 128, due to the fact that in the instructions encoding, the maximum length that the register number field can have is 7 bits. In fact, independently of the number of employed registers, in the ISA R124 is a bounded-to-zero register, normally used in operations that require a zero operand, while R127 cannot be written and is a kind of *throw-away register*, set as destination for those operations whose result is not to be saved (e.g. test or comparison instructions). These two registers are not actually accessed in FlexGrip, and the logic internal to the pipeline simulates their behavior. For configurations with a high number of threads, the threshold of 128 registers is to be kept even lower to avoid overlapping of portions between threads.
The address register files are a set of 32-bit registers as well, organized as the previous, that contains registers only used for addressing the shared memory in an indirect way (that means address register value plus offset). Special instructions allow to move values from general registers to address registers and vice versa. For each core, **128 registers times the number of lanes** are allocated, so the maximum depth is 512. Actually, just 4 address registers are assigned to each thread (A1, A2, A3 and A4).
The predicate register files have the same dimension and organization as the address registers. Locations assigned to each thread are 4 as well (C0, C1, C2 and C3) and are just 4-bit wide. These registers can save the *flags* coming out from the ALU (if the programmer specifies it in the instruction encoding), and subsequently they can be read for deciding the execution or the non-execution of *conditional instructions.* Flags are (in order): *overflow, carry, sign* and *zero.*
Address busses for both address and predicate register files are organized as follows.

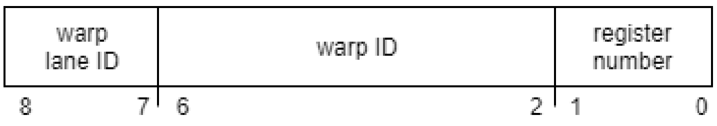


Figure 2.9: Address and predicate register address composition.

Remember that address and data busses out from Read and Write stage are multiplied by the number of cores, allowing all threads in the same warp lane to complete their reading and writing operations simultaneously.
This is not true for other portions of memory (shared, global, constant), towards which addresses are forwarded thread after thread. Thus, accesses to this memories have a larger latency and considerably impact the performance.
A main merit of the Read stage is the presence of internal *memory controllers* with adders and shifters to compute the address without using the Execute stage modules, unlike the MIPS pipeline.

2.2.4 Execution

The Execution stage is the crux of the FlexGrip pipeline, because it contains the scalar processors with all the logic and arithmetic modules necessary to perform in parallel the operations supported by the ISA. FlexGrip is an integer GPGPU, so integer addition, subtraction, multiplication and MAC (*multiply-and-accumulate*) are supported, with operands of 16 or 32 bits, signed or unsigned. Also arithmetic and logic shift are supported, integer-to-integer conversions (e.g., signed on 16 bits to signed on 32), comparisons and bitwise boolean functions (AND, OR, XOR, NOT). The result is analyzed and predicate flags are computed: if an overflow has occurred, if a carry has been produced by the addition, if the result is with sign or if it is zero.
Alongside SPs, a *flow control unit* is present, to deal with branches, synchronization point settings,

block barrier points and kernel return instructions. This module has a direct connection with the *warp stack*, a portion of LIFO outside the stage and dedicate for each warp. Let us focus first on the branching mechanisms.
Every time a conditional instruction is executed, being it a conditional branch or a normal instruction with conditional field, the Read stage retrieves the flags from the indicated predicate register, and computes an *instruction mask* which says which are the threads in the warp enabled to its execution according to the value of the flags. For example, one condition could be *equal to zero*, so the zero flag is read and only the thread with that flag set are enabled. The instruction mask has a life that ends at the Write stage: actually every SP executes the instruction but only the enabled threads write their destination locations. If the instruction is a branch, the flow control unit updates the program counter with the target address, transforms this instruction mask into the next current mask and pushes onto the warp stack the threads which are not taking the branch. The warp stack accepts words of 66 bits, organized as follows.

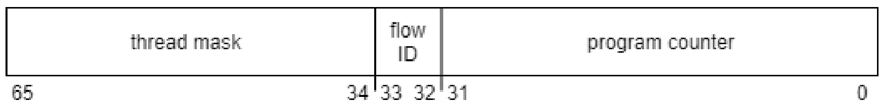


Figure 2.10: Stack word fields.

In case of branch, that fields are filled with the flow ID of the branch ("01"), the thread mask of the threads not taking the branch (computed as a bitwise AND between the current mask and the complement of the instruction mask) and the starting program counter for that group. In fact, the pipeline executes one branch after the other: once executed one of them, the stack is popped and another branch is executed, by setting mask and program counter as the ones popped out. This goes on until the *reconvergence point* is reached by all threads.
In fact, the warp branch mechanism does not work if a synchronization point is not set before the potentially divergence instructions as the branch. Even if this is not visible at high level of programming, at low level the compiler introduces a particular instructions which pushes onto the stack the information relative to the synchronization point of all the threads. The thread mask pushed is then composed by the mask of the current set of active threads within the warp (that potentially take the branch), the flow ID of the synchronization ("00") and the program counter of the synchronization instruction. This can be *any* instruction, provided that its marker indicates that it is a JOIN instruction. Every time such an instruction is executed, it means that one of the multiple branches created has reached the reconvergence point. So, the stack is popped and another branch is resumed. The last subset of threads that reaches the reconvergence point pops from the stack the synchronization point information itself, so the original mask and the common program counter are restored, and the program flow can continue from that point. The Figure 2.11 summarizes how the warp divergence handling works.

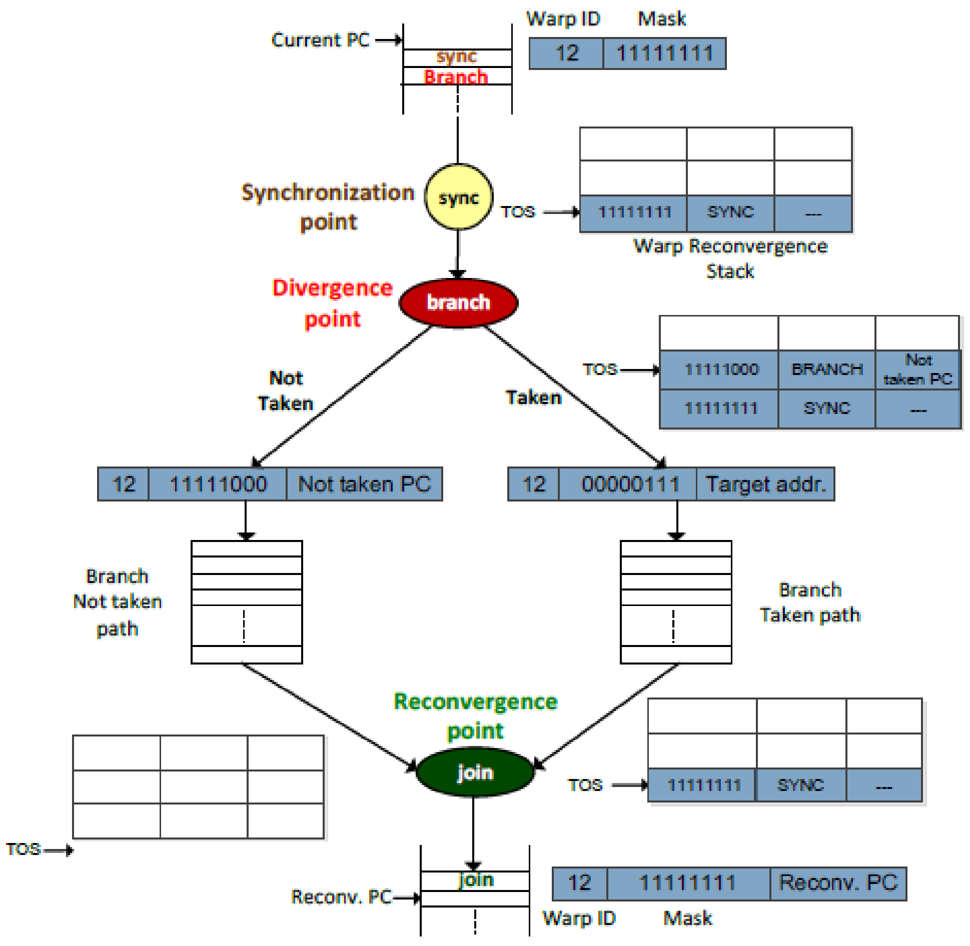


Figure 2.11: The warp divergence mechanism.

Summing up, the branch divergency is articulated along these steps:

1. The synchronization point is reached - the warp stack is populated with the synchronization point information
2. The execution reaches the branch instruction - the "taken" path is executed first, while the thread mask of the "not taken" path is pushed along with the current PC
3. The first execution branch reaches the synchronization instruction - the stack is popped and the "not taken" path is loaded in the pipeline
4. The last execution branch reaches the synchronization instruction - the synchronization point information is popped and all threads resume parallel execution

The mechanism is obviously supported also for *nested* branches, i.e., branches that further divide themselves and set a synchronization point relative just to a branch path within the warp. Levels of nesting can be up to 32, as this is the depth of the warp stack.
When return or block barrier instructions are instead recognized, the flow control unit changes the current warp state from ACTIVE to FINISHED or WAITING.

2.2.5 Write of the results

The Write stage is the last stage of the pipeline. It is specular with respect to the Read stage: it contains all the controllers for register files and memories (except for the constant memory, which

obviously cannot be written). As the Read stage was attached to port A of each storage device, the Write stage controls the B port, allowing the write of results while a read is being performed.
The chain is not broken at this stage, because the warp information and state, which have traversed all the pipeline, are delivered back to the warp unit. Here a checker reads such information and updates the pool lane and the state in the dedicate memory within the unit, so that the warp has consistent values for the scheduler.
All the pipeline stages have a *stall* signal which feeds the preceding stage, which is asserted when the stage is busy and not ready to accept new data. Outputs of the stage are only updated when the stall input signal is low.

2.3 The CUDA programming model

What has been presented up to now is the general outline of a GPGPU in terms of hardware. Now the focus moves to the software model that fits this architecture. In fact, the common programming paradigm, even with SIMD or multithreading, is not applicable here. The programmer must enter in the order of ideas of writing a routine *for a single thread which is however automatically executed by a large number of threads*, without needing to allocate them in some way, because they are already supported at hardware level. The result is a new programming paradigm, which over the years has pushed the development of several extensions of the common languages to support it. The most famous extension is definitely the **CUDA platform (Compute Unified Device Architecture)** by Nvidia, designed to allow C/C++ to support GPGPU programming without need of expertise in graphic or parallel programming.
A CUDA source file (.cu) outline is similar to normal C/C++ code, with a driving main() procedure executed by the host CPU and one or more *kernels* which contain code for the GPU. The following sample code adds two integer vectors A and B of size DIM and stores the result into vector C:

```
#include <stdio.h>
#include <cuda_runtime.h>
#define DIM 64

// Kernel definition
__global__ void vectorAdd(int *A, int *B, int *C)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    C[i] = A[i] + B[i];

}

// Host main routine
int main(void)
{
    int A[DIM], B[DIM], C[DIM];
    int blocksPerGrid = 2;
    int threadsPerBlock = 32;

    // ....

    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(A, B, C);

    return 0;
}
```

The library <cuda_runtime.h> contains the definitions for the CUDA extensions to the standard C++ library, its inclusion is mandatory. From the host main routine, the kernel can be launched passing

data parameters (between ( )) and configuration parameters (between <<< >>>), i.e., the number of threads for each block and the number of blocks composing the grid. Given that configuration, the kernel is meant to be executed by 64 threads, each one performing the addition of *one* element of A with *one* element of B. The array index for the operands is found using a common formula: assuming that we are using monodimensional blocks and grid (with only x dimension), the x block index can be 0 or 1 since the grid dimension is 2, and the x thread index goes from 0 to 31 since the block dimension is 32, so the offset variable i can assume values from 0 to 63 depending on the thread position within the grid. So, all elements are mapped by the kernel.
It is also possible to specify the length (in order) of x, y and z dimensions of the blocks and the grid by using dim3() structure:

```
    vectorAdd<<<dim3(2, 1, 1), dim3(4, 4, 2)>>>(A, B, C);
```

Actually, the z dimension for the grid is fixed to 1 for this architecture.
The grid organization of threads has actually been thought for helping the programmers to give a certain order to the parallel units of computation. We know that, at SMP level, the actual parallel unit of relevance is the warp, not controllable from CUDA. Divergencies within warps are automatically managed by both the compiler (with the adding of synchronization points) and the hardware (with the warp stack mechanism explained in the previous section). When a block-level synchronization is required among threads, the programmer has to specify it by calling the primitive

```
__syncthreads();
```

This statement introduces in the compiled code a special instructions that marks a warp as WAITING and acts as a barrier at which all threads in the block must wait before any is allowed to proceed. Divergencies at grid level are not supported by any CUDA primitive, but must be handled with programming stratagems like breaking a kernel down into multiple littler kernels or declaring global variables acting as semaphores outside the kernel, like

```
__device__ int sem = 0;
```

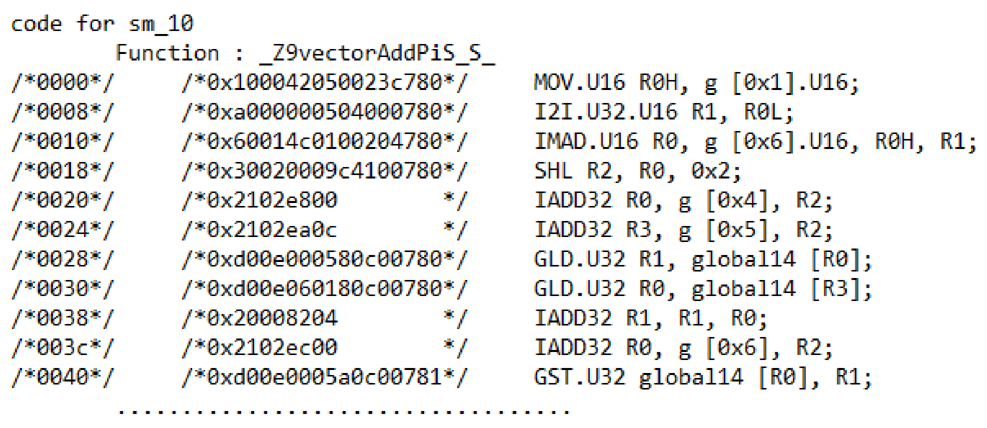
More information and hints on CUDA can be found in the numerous guides available online.

2.4 Software Flow

To obtain the binary code of the kernel necessary to feed the hardware, the CUDA source code must be submitted to the Nvidia dedicate compiler, called ***nvcc***. Such tool translates the kernel into an intermediate pseudo-assembly language which is named **PTX (Parallel Thread Execution)**. This is an open language which defines a *virtual* ISA for Nvida GPUs. Programmers can even write kernels in PTX from CUDA by using the common ANSI C directive

```
asm(...);
```

PTX makes use of virtual registers (declared as normal variables) and pseudo-instructions with clear mnemonic which are then mapped in binary through a process whose details are protected by Nvidia and not open to the programmer. This process is done at runtime by the CUDA driver API, with the production of a *.cubin* file, which through another Nvidia tool called **cuobjdump** can be disassembled in a human-readable hardware language named **SASS (Source and ASSembly code)**. SASS is specific to the target GPU architecture and represents the actual code executed on Nvida machines. A typical .sass file (e.g. the translation of the vector sum code before) appears as follows:



```
code for sm_10
        Function : _Z9vectorAddPiS_S_
/*0000*/     /*0x100042050023c780*/    MOV.U16 R0H, g [0x1].U16;
/*0008*/     /*0xa000000504000780*/    I2I.U32.U16 R1, R0L;
/*0010*/     /*0x60014c0100204780*/    IMAD.U16 R0, g [0x6].U16, R0H, R1;
/*0018*/     /*0x30020009c4100780*/    SHL R2, R0, 0x2;
/*0020*/     /*0x2102e800        */    IADD32 R0, g [0x4], R2;
/*0024*/     /*0x2102ea0c        */    IADD32 R3, g [0x5], R2;
/*0028*/     /*0xd00e000580c00780*/    GLD.U32 R1, global14 [R0];
/*0030*/     /*0xd00e060180c00780*/    GLD.U32 R0, global14 [R3];
/*0038*/     /*0x20008204        */    IADD32 R1, R1, R0;
/*003c*/     /*0x2102ec00        */    IADD32 R0, g [0x6], R2;
/*0040*/     /*0xd00e0005a0c00781*/    GST.U32 global14 [R0], R1;
        ...................................
```

Figure 2.12: Example of SASS binary code.

Each line contains the program counter, the hexadecimal encoding of the instruction and its mnemonic. The very first line indicates the *compute capability* the code is written for. With this label, Nvidia catalogs its machines in base of the hardware capabilities of each generation of GPUs. FlexGrip has been developed to support compute capability 1.0, the same as G80 first generation of graphic processors. The corresponding label is in fact *sm_ 10*, which needs also to be included as *-arch* compile option for nvcc.
This level of compute capability fixes some hardware limitations which are compliant with the current design of FlexGrip.

Warp size	32
Minimum thread size per block	32
Maximum number of thread blocks	8
Maximum dimensionality of thread blocks	3
Maximum dimensionality of blocks grid	2
Maximum number of warps per streaming multiprocessor	32
Maximum number of threads per streaming multiprocessor	1024
Number of 32-bit general-purpose register per streaming multiprocessor	16384
Shared memory size per streaming multiprocessor	16 KB
Constant memory size	8 KB
Global memory size	256 KB
System memory size	256 KB

Besides the compute capability limitations, it is also important to remember that the current version of FlexGrip

- Does not support floating-point operations
- Does not support integer division
- Does not support operations with operands wider than 32 bits
- Does not support a thread-dedicate local memory
- Does not support instructions for calling subroutines
- Does not support instructions for loop breaking
- Does not support texture memory and texture fetch.

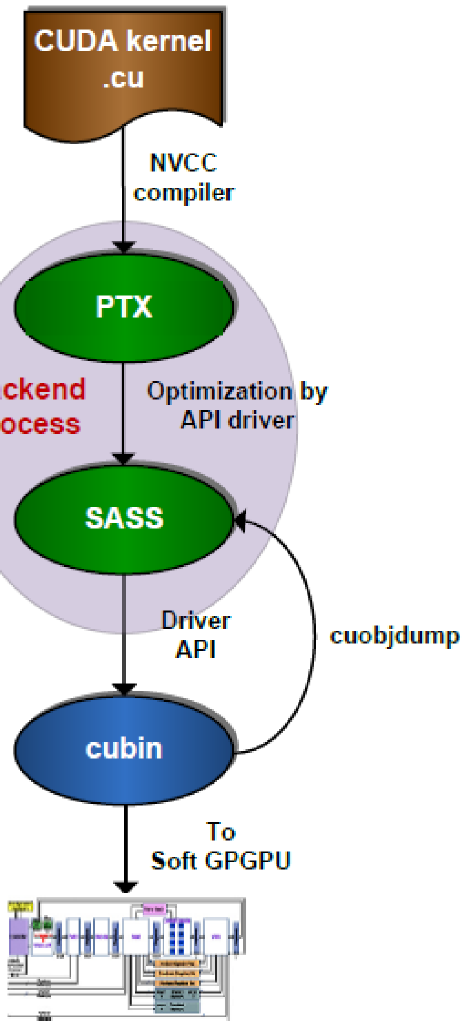


Figure 2.13: The software flow for CUDA kernels.

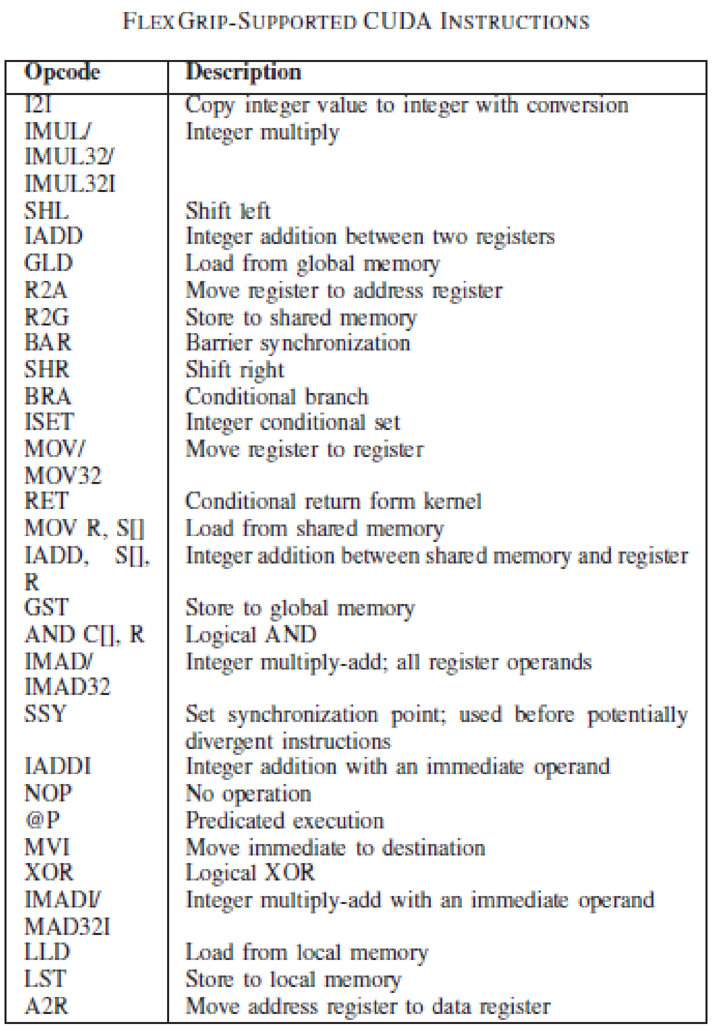
Chapter 3

Performed Activities

What has been written up to this point is the basic knowledge that was acquired *before* starting to work with FlexGrip, and that anyone else who wants to spend time on this project should have, too. This chapter will instead present the work that has been done for advancing the status of the project. At first, the model has been acquired and validated in its fundamental features. Then, the focus has been moved to the instruction level where corrections and extensions have been done. Last, 2 typical parallel applications have been developed as benchmarks of the new model for performance analysis.

3.1 Validation of the model

At first, the RTL code of FlexGrip has been analyzed to understand the hierarchy of the different modules. At this aim, a block diagram has been drawn, with the help of Simulink tool by MATLAB. The diagram is not simulable, and just helps to wrap blocks one inside the other to clarify the architecture. This first part allowed to recognize some unused ports or signals that were object of first dummy modifications of the code. Once produced the diagram, some simulations with random code already provided by the original testbench were launched in order to understand and validate the general behavior of the main modules, as the block scheduler, the streaming multiprocessor controller, the stages of the pipeline, the warp unit and the memory blocks.
A list of nominally supported instructions was included in the presentation paper of FlexGrip [1] (see Figure 3.1), and so the focus has been moved to their validation.



FLEXGRIP-SUPPORTED CUDA INSTRUCTIONS

Opcode	Description
I2I	Copy integer value to integer with conversion
IMUL/ IMUL32/ IMUL32I	Integer multiply
SHL	Shift left
IADD	Integer addition between two registers
GLD	Load from global memory
R2A	Move register to address register
R2G	Store to shared memory
BAR	Barrier synchronization
SHR	Shift right
BRA	Conditional branch
ISET	Integer conditional set
MOV/ MOV32	Move register to register
RET	Conditional return form kernel
MOV R, S[]	Load from shared memory
IADD, S[], R	Integer addition between shared memory and register
GST	Store to global memory
AND C[], R	Logical AND
IMAD/ IMAD32	Integer multiply-add; all register operands
SSY	Set synchronization point; used before potentially divergent instructions
IADDI	Integer addition with an immediate operand
NOP	No operation
@P	Predicated execution
MVI	Move immediate to destination
XOR	Logical XOR
IMADI/ MAD32I	Integer multiply-add with an immediate operand
LLD	Load from local memory
LST	Store to local memory
A2R	Move address register to data register

Figure 3.1: The initial list of instructions nominally supported by FlexGrip.

The list is *incorrect*, because some instructions, even if the description could be right, are not existing in that form (e.g. IADDI, MAD32I, AND, XOR are not instructions, while IADD32I, IMAD32I, LOP.AND, LOP.XOR are). However, it still offers an idea on what are the possibilities of the hardware in terms of arithmetic and logic operations and in term of movements between memories.
The goal was first to create some code in order to push out those instructions in compiled kernels, so the software flow was assimilated and the first dummy SASS code was produced from simple CUDA programs. Writing kernels directly in PTX has been considered as an option to produce test kernels for the instructions, so the PTX syntax has been studied and applied. Anyway, the Nvidia compiler optimizes in a deep way the code from PTX to SASS, preventing the programmer to use, for example, all the desired register numbers or all the custom format options for a given operation. A solution to this problem has been found in **compiling a huge number of random kernels directly from CUDA to push out most of the SASS instructions and formats**. This large amount of instructions has been collected and grouped by name at first: all GLD, all MOV, all IADD instructions, and so on. A *bitwise comparison* by inspection has been done between the instructions with the same name but with different operands or options in order to individuate the position of the binary fields for source, destination, operand sizes and similar. Remember that Nvidia consider the SASS code as a private resource, so never released any reference on this. The process has also been aided by the interpretation of the RTL code of the Decode stage, a pretty intricate module. The complexity in reading such uncommented and unreferenced VHDL code, plus the amount of time spent to compile

pseudo-random kernels to collect the binaries, made this process take the most significant portion of the thesis work period, about 4 months.
Anyway, every time an instruction was understood in its fundamental details, it was added to a rudimental tool, refined afterwards, which is able to **directly write a SASS file** with the instructions of a possible kernel. When a relevant number of formats for a given instruction was found, including all the memory types supported and all the possible options, a *functional test kernel* was written for it. A custom unified group of simple tools has been developed in order to:

- initialize the global memory with some values
- write a file with the golden reference of the expected results
- write down in a .sass file the test kernel containing all the formats of the instruction
- simulate the kernel
- retrieve the global memory results and compare them with the expected ones

At this point, the solutions were 2: either the match was 100% and so the process moved to another instruction, or the fixing phase started. This consisted in a graphical simulation (with waveforms) of the kernel to individuate the fault responsible of such inconsistency. Once found, the VHDL code was analyzed in order to understand what behavioral or structural statements were causing the fault, and possibly some lines of code were modified, added or removed. The test kernel was at this point relaunched and the process possibly repeated.

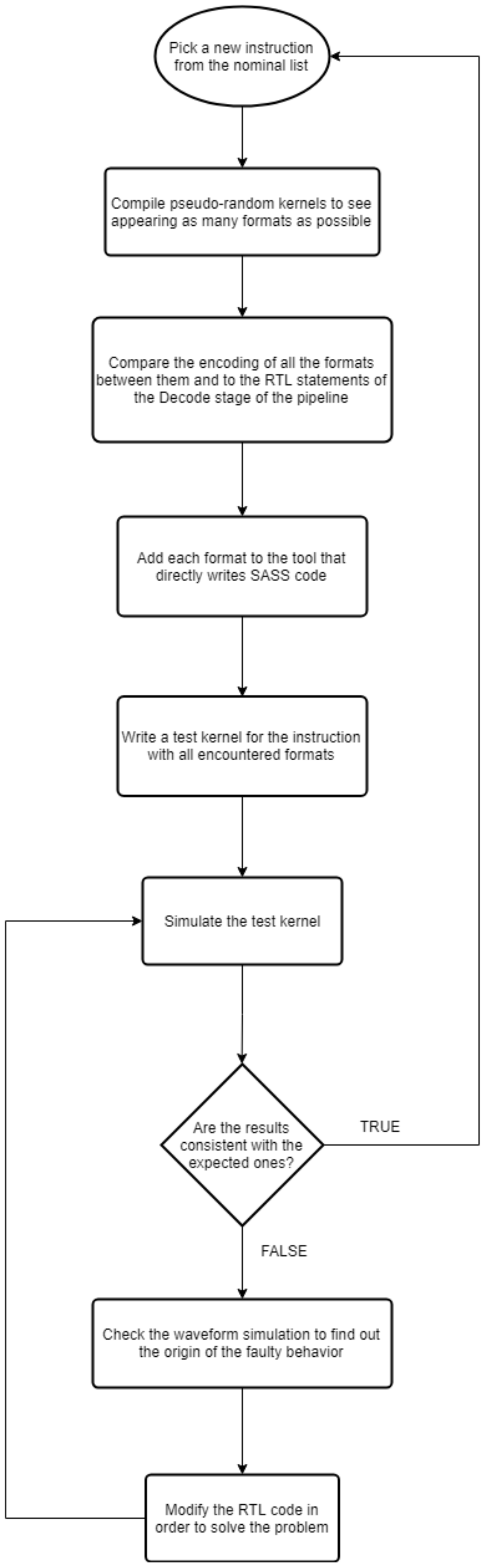

Pick a new instruction from the nominal list
Compile pseudo-random kernels to see appearing as many formats as possible
Compare the encoding of all the formats between them and to the RTL statements of the Decode stage of the pipeline
Add each format to the tool that directly writes SASS code
Write a test kernel for the instruction with all encountered formats
Simulate the test kernel
Are the results consistent with the expected ones?
TRUE
FALSE
Check the waveform simulation to find out the origin of the faulty behavior
Modify the RTL code in order to solve the problem

Figure 3.2: The instruction set validation process.

3.2 Correction and extension of the model

Even if some experiments were already made on FlexGrip to evaluate the reachable level of speedup in executing some common benchmarks with respect to non-graphic processors, the RTL code was containing a relevant percentage of *bugs*, discovered through the process explained in the previous section. Since the beginning of the thesis work, 21 of 54 VHDL source files composing the project have modified, (38.8%). Actually, this is not the percentage of modification of the whole project code, because the average of changed lines per file is 12%, so we obtain an overall amount of modification around 4.8%. Every time a problem was encountered and fixed, a new version of the project was committed to the repository. Modifications made to the code of FlexGrip can be divided into 3 categories:

1. Removals of unused ports and signals or useless statements
2. Modifications in order to make the various instruction formats working (the most relevant part)
3. Modifications for future improvement of the architecture.

The modification type at point n. 1 was the set of changes made in the very first part, even before the simulations of the kernels, because these were design imperfections visible even without need of simulating the hardware. Point n. 2 has been the most relevant part of the thesis work, while point n. 3 has arrived last and has concerned the possibility of FlexGrip to even increase its parallel computation capabilities, as it will be explained in the dedicate section in Chapter 4.
Commissions have been in total 27, 15 of which regarding the Decode stage. The module holds the record of modifications with 132 changed lines, even if for its dimensions its edit percentage is under the 15%. A summary of the modifications of the modules is reported.

Module	**Edit percentage**	**Edit description**
Decode stage	14.6%	Fixed positions of bits indicating the correct memory type, the sign, the source address and the destination address for many arithmetic and movement instructions, masking of the indicators contained in the high part of the instruction when an immediate operand is supported, adding of missing cases in some statements for particular formats, removal of some name mistakes, removal of bit assignments mismatched with the binary
Integer to integer converter within the scalar processor	60.9%	The module was not supporting all the possible integer formats and even the supported ones were converted with errors
Predicate flags calculator within the Write stage	41%	Removal of useless redefinitions and correction of the flag value decision in base of the ALU outputs
Fetch stage	30.4%	Changed the instruction bus width from 8 bit to 32 in order to use just 1 port of the dual-port system memory instead of 2 in parallel
Scalar processor	19.3%	Added a port to bring in the instruction marker and redefinitions of some signals for feeding the arithmetic modules
System memory controller	14.5%	Changed data width from 8 bits to 32 bits
Vector register file controller	14.2%	Added support for single-byte reading and writing
Adder/subtracter within the scalar processor	13.9%	Solved problems in computation of the carry and the overflow

Address calculator within the Read stage	11.4%	Shifting of the addresses was performed in the wrong direction
Execute stage	8.7%	Fixed out-of-time input sampling, redefinition of the instruction flags depending on scalar processor outputs
Data type decoder within the Read stage	8.3%	Decoding was inconsistent with the association between the mnemonic of the instruction and its binary encoding
Global memory controller	6%	Sign-extension of signed operand was not implemented, added single-byte support
GPGPU top level entity	5.5%	Modified architecture to support new width of system memory (32 bits)
Warp scheduler	3.3%	Fixed misbehaviors causing endless loops in the state machine when scheduling a FINISHED or WAITING warp
Source operand reader within the Read stage	3%	Added support for constant memory load, address-to-general-purpose register movement and fixed a bug for which only the 6 low bits of the immediate operand were read
Warp unit	1.8%	Removed unused ports
Streaming multiprocessor	1.6%	Changed system memory interface to support new width of 32 bits
GPGPU main definition package	1.6%	Adapted components port outline to modifications made and added new definition for system memory bus width (32 bits)
Write stage	1.3%	Removed useless assignments to internal signals
Read stage	0.7%	Fixed wrong instruction mask computing when popping the stack during execution of a join instruction
Streaming multiprocessor controller	0.7%	Fixed a bug for which the write enable out for the general-purpose register was hold active in states of the FSM where was not needed anymore

3.3 Verified ISA for the model

As already mentioned, the instruction validation process has been long and complicate for the absence of any documentation on the actual hardware ISA supported by Nvidia G80 family. The relevant number of different formats come out from the compilations and the analysis of the RTL description of the Decode stage were the only references available for completing the work. At the end of this process, thought, the general idea on how instructions are encoded in SASS has become surely clearer than before, and this section has the aim to present the discoveries made about it.
The set is composed by both *full* and *half* instructions, i.e., either on 64 bits or on 32 bits only. All the half instructions contains the suffix "32" in the name. The low 32 bits host the operative code of the instruction, the sources and the destination (register numbers or addresses) and some configuration bits. In case of flow instruction, the target PC of the operation is also encoded. The high 32 bits host the *subopcode*, the *instruction marker*, the possible main part of the 32-bit immediate operand, the conditional fields, the memory source type, the dimensions of the operands, the possible type of conversion to be applied for operands, the type of movement in case of some load/store/movement

instructions and other configuration bits.
Therefore, any 64-bit normal instruction is outlined as follows:

INSTRNAME.*OPTIONS*.CX (CX.*COND*) DEST, SRCS;

If the instruction is a flow instruction, its aspect is instead:

INSTRNAME.*OPTIONS* (CX.*COND*) TARGET;

A 32-bit instruction cannot have conditional fields and is outlined as the following:

INSTRNAME.*OPTIONS* DEST, SRCS;

Fields CX, *OPTIONS* and *COND* are not usually displayed if their value is default (i.e., no predicate register to be set, no conditions for the instruction, 32-bit operands).
CX can assume values C0, C1, C2 or C3 and indicates one of the four predicate registers available per thread. A predicate register can be both set (write out of the flags) and read (for conditional execution) by an instruction.
The field *OPTIONS* is actually a multiple field displaying all the instruction options. Most of the time size and sign of the operands are listed, unless they are 32-bit unsigned, which is the default data type. The option *.S* is common to all instructions and marks that instruction as a reconvergence (join) point for branch divergency (see Chapter 2). Some other instruction-specific options may appear here.
Operands of an instruction may be:

- General-purpose registers (from R0 to the last register alloted to any thread, at most R126). R0 contains the thread ID at kernel startup, R124 is a bounded-to-0 register and register #127 is a non-writable location indicated with the symbol o [0x7f]. When half registers are addressed, the syntax is RXH or RXL.

- Shared memory locations, whose symbolic is g [0x..]. If an indirect memory addressing is required, then the symbolic becomes g [AX+0x..]. AX can assume values A1, A2, A3 or A4 and indicates one of the four address registers available per thread.
  NOTE: the address register is always a *byte pointer*, while the immediate offset does respect the size of the operands, and the hardware left-shifts its value to point the correct byte. Employ of address registers is thus helpful when groups of misaligned bytes are to be addresses.

- Constant memory locations, whose symbolic is c [0x1] [0x..]. Here indirect addressing is not allowed.

- Global memory locations, whose symbolic is global14 [RX]. Here the only possible addressing is pure indirect and byte-pointing, and general-purpose registers are used instead of address ones.

- Immediate values, in hexadecimal format 0x...

- Address registers, only in case of movement from/to general-purpose registers.

Since this ISA is a register-memory architecture, most of the instruction can have **just one** operand directly taken from the memory, constant or shared. The global memory can only be accessed through load and store instructions. The destination can be a memory location **only** in case of store instructions, otherwise results are targeted to general-purpose registers.
Let us focus now on how instructions are encoded in binary. What is to be pointed out is that this is not fixed-bitfield ISA, so **bits can change their meaning even if occupying the same positions**. This is what actually makes the understanding of the Decode so difficult. A summary of the bit fields position and meaning for the known ISA is presented in Figure 3.3. The double indexes indicate the minimum and the maximum position of the edge of that field, since dimensions depend on the instruction. Other dedicate configuration bits can take the positions left by those field reduction.

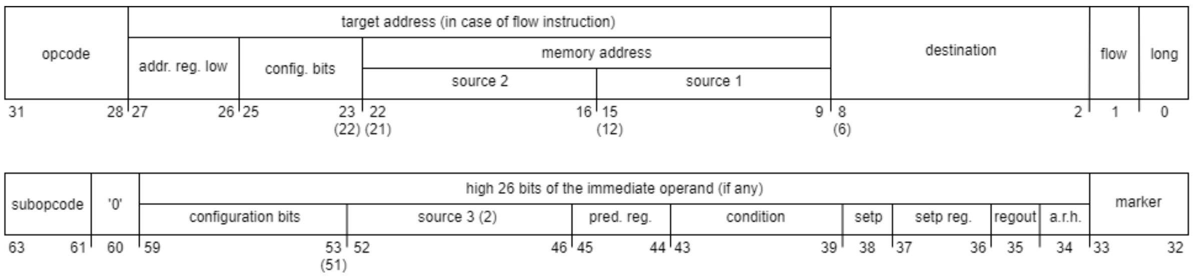


Figure 3.3: Bit fields of a typical instruction.

- *long*: this bit is set if the instruction is encoded on 64 bit, reset if on 32.
- *flow*: indicates if it is a flow instruction.
- *destination*: indicates the destination location, can be a general-purpose register or an address register at most, never a memory offset.
- *source 1*: indicates the location of the first operand, can be a memory offset or a register number; in case of instruction supporting an immediate operand, hosts from bit 14 to bit 9 the lower 6 bits of such value.
- *source 2*: indicates the location of the second operand, can be a memory offset or a register number.
- *memory address*: for some load/store instructions, indicates the source/destination memory offset.
- *target address*: indicates the next program counter for a branch instruction or a synchronization point for a reconvergence setting instruction.
- *address register (low part)*: represents the two lower bits of the address register number to be read for indirect memory addressing; if "00", no address register is added to the offset.
- *opcode*: together with the *subopcode* and other configuration fields, identifies the instruction.
- *marker*: if "00", the instruction is a normal instruction (FULL_NORM); if "01", the instruction is the last instruction of the kernel (FULL_END); if "10", the instruction has been set as a reconvergence point by a previous synchronization instruction, so the stack is to be popped (FULL_JOIN); if "11", the instruction supports an immediate operand;
- *a.r.h. (address register high part)*: represents the MSB of the address register (set to 1 only if it is A4);
- *regout*: says if the operation does or does not write the result (e.g., comparison and test instruction may not need to store the ALU result but only the flags);
- *setp (set predicate) register*: indicates the number of predicate register (C0, C1, C2 or C3) to be written with the flags produced by the operation;
- *setp (set predicate)*: says whether the flags are to be saved in a predicate register or not;
- *condition*: encodes the condition for the possible predicated execution. "01111" is the default condition and corresponds to absence of condition;
- *predicate register*: is the predicate register number to be read to see if the condition is respected and so if the instruction can be executed or not for that thread;
- *source 3 (2)*: indicates the location of the possible third operand (e.g. multiply-and-add); in some cases may indicate the second operand or also the destination;

- *subopcode*: together with the *opcode* and other configuration fields, identifies the instruction.

All other bits are used for *configuration*: they may be present or not, and assume different meaning depending on the instruction (e.g., the type of bitwise logic operation, the type of integer-to-integer conversion, the size of the operands, their source memory type, the direction of the shift and others). In the current conditions, FlexGrip supports these **28 instructions** in their overall **74 formats**:

Instruction	**Description**	**Verified Formats**
IADD	Integer addition	IADD RZ, RX, RY IADD RZ, -RX, RY IADD RZ, g [0x..], RX IADD RZ, g [0x..], -RX IADD RZ, RX, c [0x1] [0x..]
IADD32	Integer addition (half format)	IADD32 RZ, RX, RY IADD32 RZ, RX, -RY IADD32 RZ, g [0x..], RX IADD32 RZ, g [0x..], -RX IADD32.*U16* RZL|H, RXL|H, RYL|H IADD32.*U16* RZL|H, RXL|H, -RYL|H
IADD32I	Integer addition with an immediate operand	IADD32I RZ, RX, 0x.. IADD32I RZ, -RX, 0x.. IADD32I RZ, g [0x..], 0x..
I2I	Integer-to-integer conversion	I2I.*U32.U16* RZ, RXL|H I2I.*S32.S16* RZ, RXL|H I2I.*U32.S32* RZ, |RX| [1] I2I.*S32.S32* RZ, -RX I2I.*U32.U16* RZ, g [0x..].*U16* I2I.*S32.S16* RZ, g [0x..].*S16* I2I.*U32.U16.BEXT* RZ, RXL|H [2] I2I.*S32.S16.BEXT* RZ, RXL|H I2I.*U32.U16.BEXT* RZ, g [0x..].*U8* I2I.*S32.S16.BEXT* RZ, g [0x..].*S8*
IMUL	Integer multiplication	IMUL.*U16.U16* RZ, RXL|H, RYL|H IMUL.*U16.U16* RZ, g [0x..].*U16*, RXL|H IMUL.*S16.S16* RZ, RXL|H, RYL|H IMUL.*S16.S16* RZ, g [0x..].*S16*, RXL|H
IMUL32	Integer multiplication (half format)	IMUL32.*U16.U16* RZ, RXL|H, RYL|H IMUL32.*U16.U16* RZ, g [0x..].*U16*, RXL|H
IMUL32I	Integer multiplication with immediate operand	IMUL32I.*U16.U16* RZ, RXL|H, 0x.. IMUL32I.*S16.S16* RZ, RXL|H, 0x..
SHL	Shift left	SHL RZ, RX, 0x... SHL RZ, RX, RY SHL.*U16* RZL|H, RXL|H, 0x.. SHL RZ, g [0x..], 0x..

[1] for absolute value computation

[2] BEXT means Byte EXTension: operations are performed on the least significant byte of the half register

SHR	Shift right	SHR RZ, RX, 0x..
		SHR.*S32* RZ, RX, 0x..
		SHR RZ, RX, RY
		SHR.*S32* RZ, RX, RY
		SHR.*U16* RZL|H, RXL|H, 0x..
		SHR.*S16* RZL|H, RXL|H, 0x..
		SHR RZ, g [0x..], 0x..
		SHR.*S32* RZ, g [0x..], 0x..
IMAD	Integer multiply-and-add	IMAD.*U16* RZ, RXL|H, RYL|H, RW
		IMAD.*S16* RZ, RXL|H, RYL|H, RW
		IMAD.*U16* RZ, RXL|H, c [0x1] [0x..], RY
		IMAD.*S16* RZ, RXL|H, c [0x1] [0x..], RY
IMAD32	Integer multiply-and-add (half format)	IMAD32.*U16* RZ, RXL|H, RYL|H, RZ [3]
IMAD32I	Integer multiply-add with immediate operand	IMAD32I.*U16* RZ, RXL|H, 0x.., RZ
		IMAD32I.*S16* RZ, RXL|H, 0x.., RZ [4]
LOP	Bitwise logical operation	LOP.*AND/OR/XOR/PASS_B* RZ, RX, RY [5]
		LOP.*AND/OR/XOR/PASS_B* RZ, g [0x..], RX
		LOP.*AND/OR/XOR/PASS_B* RZ, RX, c [0x1] [0x..]
		LOP.*U16*.*AND/OR/XOR/PASS_B* RZL|H, RXL|H, RYL|H
R2A	Move general-purpose register to address register	R2A AX, RX
MOV	Move register to register / load from shared memory	MOV RZ, RX
		MOV.*U16* RZL|H, RXL|H
		MOV RZ, g [0x..]
		MOV.*U16* RZL|H, g [0x..].*U16*
		MOV.*U16* RZL|H, g [0x..].*U8*
MOV32	Short version of the MOV	MOV32 RZ, RX
		MOV32 RZ, g [0x..]
		MOV32.*U16* RZL|H, RXL|H
MVI	Move immediate to destination	MVI RX, 0x..
ISET	Integer comparison	ISET RZ, RX, RY, *COMP_TYPE*
		ISET RZ, RX, c [0x1] [0x..] ,*COMP_TYPE*
		ISET RZ, g [0x..], RX, *COMP_TYPE*
		ISET.*S32* RZ, RX, RY, *COMP_TYPE*
		ISET.*S32* RZ, RX, c [0x1] [0x..], *COMP_TYPE*
		ISET.*S32* RZ, g [0x..], RX, *COMP_TYPE*
BRA	Branch	BRA CX.*COND* 0x..
		BRA 0x..
BAR	Block barrier synchronization	BAR.*ARV*.*WAIT* b0, 0xfff [6]

[3]source n.3 must be the same register as destination. This is the only format known for IMAD32
[4]source n.2 must be the same register as destination
[5]PASS_B is used as NOT
[6]This is the only format that the compiler ever produced when compiling the __syncthreads() primitive. Anyway, parameters and options are not read by the hardware that just blocks all threads within the same block when this instruction is executed

RET	Return form kernel	RET RET CX.*COND*
SSY	Set synchronization point (used before potential warp divergency)	SSY 0x..
NOP	No operation	NOP
A2R	Move address register to general-purpose register	A2R RX, AX
GLD	Load from global memory	GLD.*U32|U16|S16|U8|S8* RZ, global14 [RX] [7]
GST	Store to global memory	GST.*U32|U16|S16|U8|S8* global14 [RZ], RX
R2G	Store to shared memory	R2G.*U32.U32* g [0x..], RX R2G.*U16.U16* g [0x..], RXL|H R2G.*U16.U8* g [0x..], RX [8]
MVC	Load from constant memory	MVC RX, c [0x1] [0x..]

These are all the formats that a reasonable thesis work time has been able to retreive from the hundreds of kernel compilations done. However, it is very likely than other formats of these 28 instructions may exist, and such formats may work or not. Anyway, this list is granted by the writer of this document. Each one of these formats has been added to the direct SASS assembler tool that has been developed, and proved with a dedicate test kernel. It must be said (without claims by the writer) that a right combination of these formats allows to use the current hardware capabilities of the model *in toto*.
Limitation in the limitation, these 28 instructions are **NOT** the complete instruction set valid for a G80 Nvidia machine with compute capability 1.0. That ISA also has support for floating-point conversions and operations, address registers increment, subroutine calls and even transfers from an additional level of memory called *local memory* which stands between the general-purpose registers and the shared memory in the hierarchy and works as a thread-dedicate bigger portion of storage (e.g., for larger amount of temporary data like matrixes). For obvious reasons of time, the author could not directly face up to such limits of the model. However, it is possible to find in Chapter 4 a detailed report of the suggested and hoped future improvements of the project.

3.4 Development of two common target applications: Edge Detection and Fast Fourier Transform

Once the above instruction set was validated, we wanted to see some real application run on the model to evaluate the performance. In this regard, two very common applications in the field of parallel computing have been chosen: the edge-detector filter application for images and the Fast Fourier Transform (FFT) for discrete signals.

3.4.1 Edge Detection

Edge detection consists in taking an image (seen as a matrix of color values) and applying to it a filter to bring out the contours of the figures that appear in there. The edge is considered found when a

[7] Only full registers (32-bit) can be targetted, the extension, both for signed and for unsigned, is done automatically
[8] Here the least significant byte of the register is picked

discontinuity in colours respect specific mathematic parameters. By the application of such filter, the image becomes a dark image only outlining the shapes of the objects present in the original version, and can be more easily digested by, for example, a face-recognition algorithm or something similar.

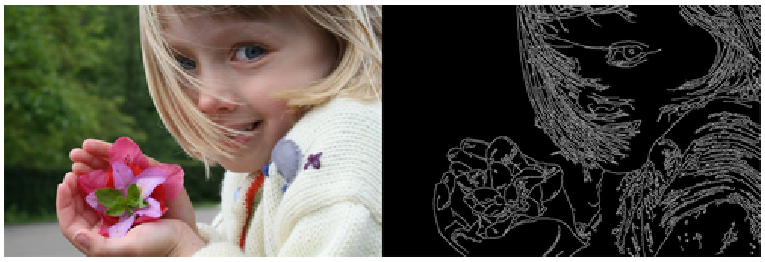


Figure 3.4: The original image and the image after the application of an edge-detection filter.

The process passes through the transformation of the picture into grayscaled unidimensional colors. If an image is encoded in RGB format, the value of the corresponding generic grayscaled pixel is

$$Y = 0.2989R + 0.5870G + 0.1140B \tag{3.1}$$

Then, the edge-detecting filter is applied through *matrix convolution* between the grayscaled image (GS) and a 3x3 matrix called *kernel*, thought for this scope, to obtain the filtered image F.

$$F = GS * \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \tag{3.2}$$

Convolution is the process of adding each element of the image to its local neighbors, weighted by the kernel. For example, if we have two 3x3 matrices, we flip both the rows and columns of the kernel and then we multiply locally similar entries and summing. The element at coordinates [2, 2] (that is, the central element) of the resulting image would be a weighted combination of all the entries of the image matrix, with weights given by the kernel:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} * \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} [2,2] = -4*e+1*f+0*i+1*h+0*g+1*d+0*a+1*b+0*c = -4e+f+h+d+b \tag{3.3}$$

Neighbors of border elements are considered null:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} * \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} [1,1] = -4*a+1*b+0*e+1*d+0*0+0*0+0*0+0*0+0*0 = -4a+b+d \tag{3.4}$$

It goes without saying that such a mole of computation is better handled by a parallel architecture as GPGPUs. The overall amount of pixels is spread among the threads, up to an upper bound of 1 pixel per thread, which operates both the grayscale conversion and the filter application. Signed integer multiplication, addition and division are needed for this algorithm. While the first two are supported by FlexGrip, the absence of the third pushed to think to alternative methods.
For the pixel/thread ratio, it was introduced the limitation of *only having powers of 2 as dimensions of the image*, and also as number of threads. Given A and B powers of 2, the division between them can be performed easily through a formula involving logarithm and left-shifts:

$$A/B = 1 << (\log(A) - \log(B)) \tag{3.5}$$

The base-2 logarithm of an integer is easily found with this algorithm:

```
int log(int n)
{
  int k = N, i = 0;
  while(k) {
    k >>= 1;
    i++;
  }
  return i - 1;
}
```

For the pixel values multiplication with decimal numbers, a mathematical reasoning has been done: multiplying for 0.2989 means multiplying for the fraction that generates such decimal number. The fraction could be any, but a fraction with a 2's power denominator is chosen in such a way that the division can be performed through a right-shift. Moreover, it must be chosen a power of 2 that guarantees the result not to lose significant digits. Since multiplications on FlexGrip are calculated on 16-bit numbers, it was considered wise to use fractions with 2^{16} (65536) as denominator. Those multiplications with the decimal numbers mentioned above have therefore been transformed as follows:

$$R \cdot 0.2989 = (R \cdot 19588) >> 16 \tag{3.6}$$

$$G \cdot 0.5870 = (G \cdot 38469) >> 16 \tag{3.7}$$

$$B \cdot 0.1140 = (B \cdot 7471) >> 16 \tag{3.8}$$

The complete source code is available in Appendix A.

3.4.2 Fast Fourier Transform

A Fast Fourier Transform algorithm is an algorithm that computes the Discrete Fourier Transform (DFT) of a sequence. This sequence is a collection of values representing samples taken from a time-continuous signal. The Fourier Transform is useful to pass from the *time domain* of a signal to the *frequency domain*, so that an evaluation of the frequency components of that signal can be done. Given a sequence of numbers $x_n, x \in \mathbb{C}$, the Fourier Transform is defined as

$$X_k = \sum_{n=0}^{N-1} x_n e^{-ik\frac{2\pi}{N}n} \qquad k = 0, \ldots, N-1 \tag{3.9}$$

Direct computation of such formula requires an $O(N^2)$ cost. Under the limitation of having N as a power of 2, the FFT is able to compute the formula with a cost $O(N \log(N))$. The most diffused FFT algorithm is the Cooley-Turkey one, which computes the element in groups of growing powers at each step/loop. To explain, at first loop elements x_0, x_2, x_4, x_6 and so on are taken for computation, and the others are excluded; the next cycle, elements x_0, x_1, x_3, x_4 and so on are taken, while x_1, x_2, x_5, x_6 are excluded, and so on. The number of cycles is equal to the logarithm of the dimension of the vector, at the last loop elements from x_0 to $x_{N/2-1}$ are taken. Before starting the computation, elements in the vector are reordered following a specific index order to make the process possible. Then, the roots of the unit, ($e^{-ik\frac{2\pi}{N}n} \quad k = 0, \ldots, N/2-1$) must be computed to be multiplied for the elements. Once done that, the actual algorithm can start.
The version of FFT developed for the project performs the ordering and the roots computation *outside* the kernel to bypass the GPU limitations. The main routine passes the ordered array and the array of the roots to the kernel function, along with the dimension. A specific data structure has been defined for the complex numbers, and since division and floating-point numbers are not supported, an artificial fixed-point system has been studied: the real and the immaginary parts of the number are 32-bit signed integers *whose 8 low bits are to be considered as the fractional part.* While the addition does not affect this scheme, the multiplication makes the fractional part double its length, taking the low 16 bits. Therefore, at the kernel return, values are to be considered as composed of *16 bits of*

integer part and 16 bits of fractional part.
The only division required is when elements are to be assigned to each thread, and a *start* and a *stop* index within the array are to be computed for each of them. At this aim, the logarithm method is employed, since both total amount of thread and number of elements are powers of 2.
It must be underlined that, contrary to the Edge Detection application, here the parallelism of the GPGPU is *half wasted*, since at any instant only half of the elements must be loaded and processed, while the threads which have been assigned to those values that are not to be considered in the present cycle are still.
The complete source code, with verbous comments to better understand each part, is available in Appendix A.

Chapter 4

Conclusions

4.1 Performance evaluation

The following tables report the execution times of the two applications developed depending on the number of parallel cores inside the SMP and the number of threads used. As input image of the Edge Detection kernel, a 16x16 pixels image has been chosen, while for the FFT a signal of ordered samples from 1 to 64 (ramp) has been taken. Since it is impossible to have a number of threads lower than the dimension of a warp (32) and greater than the dimension of the inputs (at most 1 pixel/sample per thread), the performance of the FFT has been evaluated with 32 and 64 threads, while the Edge Detection has been simulated with 32, 64, 128 and 256 parallel threads, for each value of internal hardware parallelism. The reduced sizes of the inputs allowed to more easily check at first sight the coherence of the results, anyway computed through C functions which reproduce exactly the statements of the kernel. The clock speed has been left equal to the original one of the project, 100 MHz.

FFT:

	64 threads	**32 threads**
8 cores	963.270 ns	955.850 ns
16 cores	587.150 ns	650.730 ns
32 cores	405.790 ns	494.150 ns

Edge Detection:

	256 threads	**128 threads**	**64 threads**	**32 threads**
8 cores	2.591.720 ns	1.594.460 ns	1.129.510 ns	1.069.870 ns
16 cores	1.430.610 ns	898.230 ns	638.460 ns	710.660 ns
32 cores	848.480 ns	548.420 ns	415.170 ns	527.110 ns

The times are referred to a *flat* grid configuration, i.e. with all threads occupying one line only in the block and with just one block collecting all threads. In fact, it has been observed that changing the x, y and z dimensions of the block and of the grid does not affect the execution time, and as well distributing the number of threads over more threads only produces an increase of the execution time of 460 ns each new block, which is the allocation time in the shared memory of the block by the controller.

First thing to be noticed is the time saving brought by increasing the hardware parallelism. The FFT boost its performance of its 39% when passing from 8 to 16 cores, and of its 58% when passing to 32, while the Edge Detection improves of a 45% from 8 to 16 and of a 67% from 8 to 32. The gain has a negative 2nd derivative over the number of cores due to the fact that while arithmetic and logic operations from/to registers are perfectly parallelized, loads and stores towards memory are in any case executed serially, therefore a lower horizontal asymptote is present, representing the time of execution of all the memory operations, which are far more expensive in terms of time, especially if towards the global memory.
The strange behavior is the apparently paradoxal decrease of the execution time when the number of threads is lowered in the Edge Detection kernel. This is actually provoked by the fact that *division is absent in the design*, and the relevant number of divisions needed before the actual computation for dividing the data along the threads, is to be computed following the logarithm method explained in the previous Chapter. This task is pretty time-consuming, for the presence of loops executed by all threads. However, the lower the number of threads, the lower the number of loops, and also the lower the number of parallel executions of such threads. This results in an evident performance boost. The effect is less visible in the FFT for the much lower number of divisions to be done.
As it is possibile to notice, the tendency inverts when the number of threads reaches the lower bound, in the 16-core and in the 32-core execution. Here the gain brought by the lower number of logarithms computed cannot compensate the increase of number of actual loops made by each thread for the image computation (the number of pixel-per-thread increases).
In the 8-core execution with 32 threads of the Edge Detection this fashion is not visible but the negative slope *is* lowering down in any case. In the 8-core exeuction with 32 threads of the FFT, instead, the performance are improved a little with respect to the 64-thread version, but the causing phenomenon is the same: the logarithm loops are lower, and this affects the performance more than what the grow of the actual computational cycles for each thread does.

4.2 Further developments

FlexGrip is a very useful model for GPGPU simulation and study for its pretty high accurancy and its depth of detail description. However, it still has *limits* that, in view of future developments, would be a shame not trying to overcome.
The most striking and most stringent limit from the point of view of simulative possibilities and not only is the absence of a multi-SMP environment. This means that the parallelism is only exploited at the core level, by playing with the number of internal scalar processors in the Execute stage of the pipeline. The block scheduler in the current conditions does not implement any dispatching algorithm, and just assignes the whole grid to the only SMP standing.
Anyway, the dual-ported nature of all the device memories (global, constant, system), under certain conditions, potentially supports the introduction of a second multiprocessor. In the current design, these three blocks of memory are contended (for both ports) between the signals coming from the extern and the SMP. In fact, the external driver needs to write both the instructions of the kernel in the system memory and the constants needed for its execution in the constant memory. At the end of the kernel, then, it needs to access the global memory for reading the results. For this purposes, it actually requires one port only, even if for sake of design coherence, when the signal of external control is up, both ports are accessible from the extern.
As already explained in Chapter 2, the design is made up in such a way that each portion of memory can be accessed simultaneously for reading and for writing purposes, in and out of the SMP. Anyway, the constant and the system memory are not thought to be written but by the driver at the configuration time. The result is that port B of the constant memory, when the external control is released and the kernel is running, has all signals (data, address and write enable) *bounded to 0.* Therefore, thesfe signals can be attached to the constant memory interface of the second SMP with no additional effort.
For the system memory, the situation is a little different, because originally the architect designed both the ports in control of the SMP in order to perform a fetch of 16 bits of the instruction each cycle, sending two consecutive addresses on ports A and B and reading 2 bytes at a time. The mechanism respected the original memory data width of single bytes, but it was expensive for time and utilization

overheads. My step ahead was then to extend the system memory width from byte to quadwords (32 bits) in order to boost the fetch process (maximum fetch time has been lowered from 4 cycles to 2) and especially to leave one port free and available for the incoming second multiprocessor.
Unlike these two storage blocks, the global memory is a read-write memory, so once the driver releases the control, port A is bounded to Read stage internal to the pipeline of the SMP while port B is bounded to the Write stage. The insertion of a second multiprocessor inevitably leads to introduce a mechanism of *arbitration* for the access. Such a mechanism is already present within the streaming multiprocessor in different zones, like for example in the Read stage, where the 3 different source-gathering modules may want to access the same memory block while the interface is just one. The arbiter present here could be replicated at the extern of the SMPs and could decide for global memory contention.
There is also a fourth space of storage outside the multiprocessors, that is the configuration register file. This module is written at configuration time by the external drivers, and reflects the value of its registers directly outside to the block scheduler, but it is also accessed by a normal register file by the SMP controller at kernel startup time. Here the modification suggested is straightforward: the block could be redesigned as a dual-port register file as all the other ones present in the model, so that both SMPs can access it.
Once solved the contention for the common external memories, still the block scheduler needs to be modified to support a two-multiprocessor environment. The ports of the module are to be minimally changed: the signal *smp_done_in* is to be extended from single to double line to receive both the done signals from both the SMP. As well, *block_idx_out* on 16 bits is to be doubled for indicating to each SMP the number of the first block assigned to it. Internally, an algorithm for correctly partition the total number of blocks should be implemented. For example, since the grid dimensions are a power of 2 and so the blocks can be 1, 2, 4 or 8 at most, if just one block is to be scheduled then it is assigned to one of the 2 SMPs randomly, otherwise blocks are divided equally over the multiprocessors. The output interface of the scheduler, apart from the block index signal, can remain untouched, because those values are equal for all the SMPs present, so they can feed both multiprocessors. Also the SMP enable signal (activated at the end of the scheduling) and the SMP reset (delivered at the end of the kernel, when all SMPs are done) can feed both the SMPs.
With all these modification, a two-multiprocessor GPGPU could be simulated. For designs that contemplate the presence of more than 2 SMPs, the mechanism of arbitration could be adopted also for constant memory, system memory and configuration registers, while the block scheduler could only be adapted to the new number in terms of interface and internal policy. If smart arbiters were placed alongside memory ports and an intelligent and parametric policy was implemented inside the block scheduler, making the number of multiprocessor a generic parameter for the design would be not so difficult.
The other big limit of FlexGrip is its partial support for the complete ISA of the Nvidia G80 family, at least for compute capability 1.0. It would be really positive to introduce a floating-point unit, even because in some compilations among the hundreds done since the beginning of this work, arithmetic instructions like FADD or FMUL or conversion instructions like F2I, I2F or F2F have appeared. The thesis time was insufficient to think about the FPU insertion, however it is not so difficult to find and adapt something already done online. If an FPU were added to the scalar processor, it would then be just a matter of writing the necessary statements for the floating-point instructions into the Decode stage (once understood the instruction encoding) and that is it.
The original G80 ISA also contemplates the presence of an intermediate level of immediate memory between the registers and the shared memory, which is the **local memory**. Loads and stores from/to this thread-reserved portion appear for example when compiling kernels which make use of relevant quantities of intermediate memory (e.g., a temporary variable where to store an array or a matrix) that the register file is unable to fit but that are anyway reserved to a single thread. The instructions LLD and LST were also already present in the original list of nominally supported instructions, anyway in the current conditions there is no local memory, and neither is the hardware equipped to support it. It is therefore a question of rethinking the Read and Write stages in a deep way in order to substain this further level of memory, which can be very useful for some applications.
Instead, there seems to be support for some flow control instructions (such as subroutine call CAL or conditional break from loop BRK) in the dedicate unit within the pipeline execute. However, since

these instructions were not belonging to the nominally supported list, the investigation of the present work has not gone beyond the realization of this fact.
As for the division, it seems that it is not contemplated at the level of compute capability implemented by FlexGrip, and even the compiler rephrases a division with tricks involving multiplication and shifts. Here the *non plus ultra* would be fabricating from scratch the opcode and the instruction encoding for the integer division at least, and adapting a dedicate unit within the scalar processor for its execution.

Appendix A

CUDA source code for the two developed benchmarks

The code for the benchmarks developed for evaluating the performance of the design is reported here.

EdgeDetection.cu:

```
typedef struct {
  uint8_t R;
  uint8_t G;
  uint8_t B;
} pixel;


__device__ uint8_t img_grayscaled[DIM1][DIM2]; // intermediate grayscaled
    image
// IMPORTANT NOTE: the compiler takes into account that the address of
   this global variable is stored in the constant memory at address c [0
   xe] [0x0].
// This zone is unknown for FlexGrip, so to avoid problems, after
   compilation all the instructions containing that address in the SASS
   are to be changed to c [0x1] [0x2].


// Edge Detection kernel
__global__ void EdgeDetection(pixel imgin[][DIM2], uint8_t imgout[][DIM2
   ], size_t M, size_t N)
{
  // global thread ID computation
  int a = blockDim.x;
  int b = blockDim.y;
  int w = blockDim.z;
  int A = gridDim.x;
  int B = gridDim.y;
  int x = threadIdx.x;
  int y = threadIdx.y;
  int z = threadIdx.z;
  int X = blockIdx.x;
  int Y = blockIdx.y;
  int ops = x + (y*a) + (z*a*b) + (X*a*b*w) + (Y*a*b*w*A);
```

```
// image is MxN pixels, divided into as many subimages as the threads
    are
// each thread takes as many pixels as it can in a single row
// if pixels per thread (ppt) number is lower than pixels in a line (N)
    , then more than one thread occupies a line, and each subimage has m
    =1 and n=N/tpl (threads per line)
// if ppt is equal to N, then just one thread occupies a line, and each
     subimage has m=1 and n=N
// if ppt is greater than N, then more than 1 line is assigned to a
    single thread, and each subimage has m=M/lpt (lines per thread) and
    n=N
//  _____________________________________
// |_________________|__________________|  ^
// |_________________|__________________|  |
// |_________________|__________________| M
// |_________________|__________________|  |
// |_________________|__________________|  ^
// |_________________|__________________|
//                  <- N ->

int m; // height of subimage
int n; // width of subimage
int r, c; // starting pixel coordinates
int i, j; // cursors for moving through the pixels
int16_t s; // temporary accumulator for computation
int totPix = M*N; // total number of pixels
int totThreads = a*b*w*A*B; // total number of threads
int ppt; // pixels-per-thread
int tpl; // threads-per-line (not used if ppt > N)
int lpt; // lines-per-thread (not used if ppt <= N)

// unfortunately division is not supported by FlexGrip architecture,
    but since divisions are between powers of 2, we can use the
    logarithm method
// ppt computation
int k1 = totPix;
int k2 = totThreads;
int lg1 = 0;
int lg2 = 0;
while(k1 > 0) {
    k1 = k1 >> 1;
    lg1++;
  }
lg1--;
while(k2 > 0) {
  k2 = k2 >> 1;
  lg2++;
}
lg2--;
ppt = 1 << (lg1 - lg2);

if(ppt <= N) {
  // tpl computation
  k1 = N;
```

```
    k2 = ppt;
    lg1 = 0;
    lg2 = 0;
    while(k1 > 0) {
      k1 = k1 >> 1;
      lg1++;
    }
    lg1--;
    while(k2 > 0) {
      k2 = k2 >> 1;
      lg2++;
    }
    lg2--;
    tpl = 1 << (lg1 - lg2);
    // starting row index is found as ops shifted right of the logarithm
       of the number of threads contained in a line
    k1 = tpl;
    lg1 = 0;
    while(k1 > 0) {
      k1 = k1 >> 1;
      lg1++;
    }
    lg1--;
    r = ops >> lg1;
    // starting column index is found as ppt*(ops MOD tpl), i.e., ppt*(
       ops AND (tpl-1))
    c = ppt*(ops & (tpl-1));
    // number of rows of the subimage is just 1
    m = 1;
    // number of columns of the subimage is N/tpl (possibly all columns
       if ppt == N)
    k1 = N;
    k2 = tpl;
    lg1 = 0;
    lg2 = 0;
    while(k1 > 0) {
      k1 = k1 >> 1;
      lg1++;
    }
    lg1--;
    while(k2 > 0) {
      k2 = k2 >> 1;
      lg2++;
    }
    lg2--;
    n = 1 << (lg1 - lg2);
  }
  else {
    // lpt computation
    k1 = ppt;
    k2 = N;
    lg1 = 0;
    lg2 = 0;
    while(k1 > 0) {
      k1 = k1 >> 1;
      lg1++;
    }
    lg1--;
    while(k2 > 0) {
      k2 = k2 >> 1;
      lg2++;
    }
    lg2--;
    lpt = 1 << (lg1 - lg2);
    // starting row index is found as ops shifted left of the logarithm
       of lpt
    k1 = lpt;
    lg1 = 0;
    while(k1 > 0) {
      k1 = k1 >> 1;
      lg1++;
    }
    lg1--;
    r = ops << lg1;
    // starting column index is just 0
    c = 0;
    // number of rows of subimage is given by M/lpt
    k1 = M;
    k2 = lpt;
    lg1 = 0;
    lg2 = 0;
    while(k1 > 0) {
      k1 = k1 >> 1;
      lg1++;
    }
    lg1--;
    while(k2 > 0) {
      k2 = k2 >> 1;
      lg2++;
    }
    lg2--;
    m = 1 << (lg1 - lg2);
    // number of columns of subimage: all the columns, so N
    n = N;
}

// grayscaled image computation
for(i=0; i<m; i++) {
    for(j=0; j<n; j++) {
      s = ((imgin[r+i][c+j].R*0x4C84)>>16) + ((imgin[r+i][c+j].G*0x9645)
          >>16) + ((imgin[r+i][c+j].B*0x12DF)>>16);
      if(s > 255) img_grayscaled[r+i][c+j] = 255;
      else img_grayscaled[r+i][c+j] = (uint8_t)s;
    }
}

// filter application
for(i=0; i<m; i++) {
    for(j=0; j<n; j++) {
      // clockwise sense
```

```
            s = 0;
            s = s + (img_grayscaled[r+i][c+j])*-4;
            if(c+j+1 <= N-1) s = s + img_grayscaled[r+i][c+j+1];
            if(r+i+1 <= M-1) s = s + img_grayscaled[r+i+1][c+j];
            if(c+j-1 >= 0) s = s + img_grayscaled[r+i][c+j-1];
            if(r+i-1 >= 0) s = s + img_grayscaled[r+i-1][c+j];
            if(s < 0) s = 0;
            else if(s > 255) s = 255;
            imgout[r+i][c+j] = (uint8_t)s;
        }
    }

}
```

FFT.cu

```
typedef struct {
  int32_t real;
  int32_t imm;
} complex;



int lg2(int N)    //function to calculate the logarithm base 2 of an
    integer
{
  int k = N, i = 0;
  while(k) {
    k >>= 1;
    i++;
  }
  return i - 1;
}



int reverse(int N, int n)    //calculates the reverse index of each
    number with respect to the dimension N
{
  int j, p = 0;
  for(j = 1; j <= lg2(N); j++) {
    if(n & (1 << (lg2(N) - j)))
      p |= 1 << (j - 1);
    }
  return p;
}



void ordina(complex *f1, int N)    //disposes elements of the array with
     respect to the reverse function order
{
  int i, j;
  complex *f2 = (complex*)malloc(N*sizeof(complex));
  for(i = 0; i < N; i++) {
    f2[i].real = f1[reverse(N, i)].real;
    f2[i].imm = f1[reverse(N, i)].imm;
  }

  for(j = 0; j < N; j++) {
    f1[j].real = f2[j].real;
    f1[j].imm = f2[j].imm;
  }
}


// FFT kernel
__global__ void FastFourierTransform(complex *v, complex *W, int N)
{
```

```
// global thread ID computation
int a = blockDim.x;
int b = blockDim.y;
int w = blockDim.z;
int A = gridDim.x;
int B = gridDim.y;
int x = threadIdx.x;
int y = threadIdx.y;
int z = threadIdx.z;
int X = blockIdx.x;
int Y = blockIdx.y;
int ops = x + (y*a) + (z*a*b) + (X*a*b*w) + (Y*a*b*w*A);

// input array is composed by N complex numbers, divided into as many
    subarrays as the threads are
// threads cannot be more than elements, of course

int totThreads = a*b*w*A*B; // total number of threads
int j; // counter of the main loop
int i; // counter of the inner loop
int ept; // elements per thread
int start; // starting index
int stop; // finish index
int lg2dim = 0; // logarithm of the dimension - needed for knowing how
    many times the application must cycle
int k1, lg1; // support variables for computing the logarithms
complex temp; // 1st temporary variable for FFT computing - see Cooley-
    Turkey algorithm for reference
complex Temp; // 2nd temporary variable for FFT computing - see Cooley-
    Turkey algorithm for reference
int n = 1; // see Cooley-Turkey algorithm for reference
int m = N >> 1; // see Cooley-Turkey algorithm for reference
int md = m-1; // mask for the index - see Cooley-Turkey algorithm for
    reference

// compute the logarithm of the dimension to individuate the number of
    loops to be done
k1 = N;
while(k1 > 0) {
  k1 = k1 >> 1;
  lg2dim++;
}
lg2dim--;

// compute elements per thread
k1 = totThreads;
lg1 = 0;
while(k1 > 0) {
    k1 = k1 >> 1;
    lg1++;
  }
lg1--;
ept = 1 << (lg2dim - lg1);

// compute the starting and final index in the array for each thread,
      which are respectively equal to ops*ept and (ops+1)*ept
  start = ops*ept;
  stop = (ops+1)*ept;

  // actual computation of the transformed vector
  for(j=0; j<lg2dim; j++) {
    for(i=start; i<stop; i++)  {
      if(!(i & n)) {
        // at each cycle, elements with indexes taken in groups of n in
            an alternate fashion are computed
        temp.real = v[i].real;
        temp.imm = v[i].imm;
        // complex multiplication: computed as a binomial product of real
             and imm
        // magnitude of elements of f should not overcome 2^15 not to
            incur in overflow in the multiplication
        // individuation of the index of W is translated from the
            original form (i*m)%(n*m) to simplified form (i*m)&(DIM >> 1),
        // because n*m is always DIM/2 , and MOD DIM/2 corresponds to AND
             (DIM/2)-1
        Temp.real = (W[(i*m)&(md)].real * v[i+n].real) - (W[(i*m)&(md)].
            imm * v[i+n].imm);
        Temp.imm = (W[(i*m)&(md)].imm * v[i+n].real) + (W[(i*m)&(md)].
            real * v[i+n].imm);
        /*
            ***********************************************************************
            */
        v[i].real = temp.real + Temp.real;
        v[i].imm = temp.imm + Temp.imm;
        v[i+n].real = temp.real - Temp.real;
        v[i+n].imm = temp.imm - Temp.imm;
        }
    }
    n = n << 1;
    m = m >> 1;
  }
}
```

Appendix B

How to simulate FlexGrip

This user guide has been added to the document in order to show how to set up FlexGrip and run simulations.

B.1 System requirements

1. Microsoft Windows 7
2. Mentor Graphics ModelSim 10.0 SE (or similar)
3. NVidia CUDA Toolkit 4.0 64-bit
4. Python 3.6.5

B.2 Setting up and validating the environment

First, download FlexGrip.rar from the website and unrar the data. The project has two main folders: /GenericDesign, where the VHDL code and the configuration files are contained, and /NewApplication, from where a new benchmark starting from CUDA or directly from SASS can be written.
GenericDesign at its first level contains:

- The top level entity, its components and the package with all the definitions
- The folder with the description of all the SMP components
- A folder containing all the configuration files (TCL scripts, memory initialization files) to use the environment
- The folder with all the testbench files.

In the /TB folder, 11 benchmarks can be found. 9 are the test programs used for validating each instruction family, and 2 are the applications developed by the author of this document for performance evaluation. The last application, TP ("Test Program"), is the one that the user can customize for its own kernel.
The file *pick_bench.vhd* is the testbench header and contains the parameters will be set or passed to the design. It is here defined the number of parallel scalar processors inside the SMP, the name of the application to be run, the dimensions of the block and the grid, the initialization data for constant memory and the kernel data parameters. Therefore, here the parallel configuration of FlexGrip can be changed before running the simulation to achieve the desired number of threads or internal parallel processors. The number of CORES can be 8, 16, or 32, and according to it the WARP_LANES must be set respectively to 4, 2 or 1.
The parameters BENCH_APP and BENCH_APP_INST have the same value and represent the

application name, which can be one of the 12 names of the subfolders in the TB folder. The correct application must be set as well by uncommenting the corresponding #define in the file *gpgpu_test.h*. This header file is useful for automatizing the validation process, whose list of commands is in the execution script *gpgpu_test.bat* (or *gpgpu_test.sh* if Unix). When such script is launched, the correct global memory initialization file for each application is copied into the one read by the design, and also the file containing the expected result is produced. As last command, contents of *gpgpu_rdata.log*, containing the kernel simulation results, and *expect.log*, containing the expected values, are compared and a percentage of success of the test is computed.
In the end, 4 files are to be modified if one of the 11 already present benchmarks is to be executed:

1. *pick_bench.vhd*, for setting the cores and the BENCH_APP name (the block/grid configuration is fixed into the corresponding *_configuration.vhd* file for each instruction test program, but still remains editable for FFT and EdgeDetection)

2. *gpgpu_test.h* by uncommenting the correct application name

3. *gpgpu_compile.tcl* for setting the simulation time (test programs for instructions run in less than 200 us, for the two main apps see the table at Section 4.1)

4. *gpgpu_test.sh* or *gpgpu_test.bat* for setting the BENCH_APP name as well

Then, the test script is to be launched from the path /GenericDesign/lib.
If a graphical simulation of those benchmarks is required, the TCL compiling script can be launched directly from the GUI of ModelSim/QuestaSim. In this case, the memory initialization tool is the only one to be launched. The simulation tool will show the waves listed in the file *wave.do*, at the beginning of which the correct number of cores must be set too not to lose details.
In the /NewApplication folder, the source code of any of these benchmarks is stored.

B.3 Writing custom applications

If a custom application is required, then solutions are 2: either it can be written using the CUDA language and toolkit at high level, or it can be developed at assembly level directly.
Go to /NewApplication folder. Here an example CUDA program, *TP.cu*, is shown. Please see any reference on CUDA syntax and programming model. Modify such file with the customized app and launch the *cu_compile.bat* script. At first, it invokes the compilation makefile present in the folder. The .cu file is given to the *nvcc* tool. A compiled *.cubin* binary file is produced, and a further command *cuobjdump* translates such binary into the human-readable SASS language. Also the PTX version of the kernel is produced. Then, a Python script is called to "hardwire" the SASS into the VHDL files read by the testbench for configurations parameters and instructions. *cu_compile.bat* as last operation copies these 2 files into the /GenericDesign/TB/TP folder. At this point, change the application name in *pick_bench.vhd*, possibly produce your memory initialization file and do not forget to change constants and kernel parameters from the *_regs_default* arrays in *pick_bench.vhd*. The TP app in fact reads from those arrays the initial state of the shared (kernel parameters) and the constant memory. The programmer must be careful and literally guess what are the constants used in the program. Arrays are in descendent order of index, so start from the last element to insert them. Unlike parameters, constant memory is byte-organized even if it is accessed with 32-bit movements.
Once done this, from /GenericDesign/lib launch the simulation tool and the TCL script within its GUI and that is it.
If instead you want to write a kernel directly using SASS language, open the *sass_assembler.c* file. At the bottom of the file you can insert one after the other the calls to functions referred to any working machine format. This program does nothing else than creating a .sass file and writing into it one after the other the instructions according to the parameters passed to procedures. It adds by default a RET instruction at the end and a NOP after it. Once written the list of instructions, from this location launch the script *sass_assembly* (present in Linux Bash version or Windows Batch). The script compiles and executes the assembler, and then translates into VHDL files the SASS kernel and copies into /GenericDesign/TB/TP folder. At this point, in order to simulate it, the steps to be followed are

identical to the previous case.

NOTE: Unfortunately, Nvidia Toolkit is only released for Windows. For compiling CUDA programs, a Windows environment is required. If you are working on a Linux distribution, you have to manually copy the *_configuration.vhd* and the *_instructions.vhd* files into the destination folder for the simulation after having switched from Windows to Linux.

Bibliography

[1] Kevin Andryc, Murtaza Merchant, and Russell Tessier, *FlexGrip: A Soft GPGPU for FPGAs*, Department of Electrical and Computer Engineering University of Massachusetts, Amherst, MA, USA, December 2013.

[2] Kevin Andryc, Murtaza Merchant, and Russell Tessier, *Soft GPGPUs for Embedded FPGAs: An Architectural Evaluation*, Department of Electrical and Computer Engineering University of Massachusetts, Amherst, MA, USA, June 2016.

[3] Martin Dimitrov, Mike Mantor, Huiyang Zhou, *Understanding Software Approaches for GPGPU Reliability*, University of Central Florida, Orlando, March 2009.

[4] Nvidia Corporation, *Nvidia CUDA C Programming Guide version 4.2*, April 2012.

[5] Nvidia Corporation, *Nvidia Compute - PTX: Parallel Thread Execution ISA version 1.4*, March 2009.

[6] Sunpyo Hong, Hyesoon Kim *An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness*, Georgia Institute of Technology, Atlanta, GA, USA, 2009.

[7] B. Du, Josie E. Rodriguez Condia, M. Sonza Reorda, L. Sterpone *About the functional test of the GPGPU scheduler*, Politecnico di Torino, Torino, Italy, 2018.

[8] Murtaza Merchant, *Testing and Validation of a Prototype GPGPU Design for FPGAs*, University of Massachusetts Amherst, February 2014.

[9] http://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units

[10] http://en.wikipedia.org/wiki/CUDA

[11] http://en.wikipedia.org/wiki/Edge_detection

[12] http://en.wikipedia.org/wiki/Kernel_(image_processing)

[13] http://en.wikipedia.org/wiki/Fast_Fourier_transform