Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

GPU-BASED MANY-CORE ARCHITECTURE EMULATION: A DOUBLE LEVEL APPROACH

Tesi di Laurea in Sistemi Virtuali

Relatore:
Chiar.mo Prof.
Renzo Davoli

Presentata da:
Andrea Monzali

Correlatore:
Chiar.mo Prof.
Moreno Marzolla

Sessione II
Anno Accademico 2012/2013

“Con il simulatore di volo non si va in vacanza”

“Roooawh”
(Chewbacca)

Introduzione

Già nel 1965 Gordon Moore (co-fondatore di Intel) teorizzò che il crescente numero di transistor presenti in un microprocessore e la loro sempre più piccola dimensione, avrebbero portato nei primi anni 2000 ad un'attenuazione nella crescita di tali dispositivi. Per andare incontro a questo fatto, la comunità scientifica ha cominciato già da diversi anni ad indagare il calcolo parallelo: se non è più possibile aumentare la potenza dei processori, è possibile introdurre più unità di calcolo in un unico computer, delineando i due modelli di calcolo parallelo *multi-core* e *many-core*. Il primo è attualmente disponibile nella maggior parte dei PC acquistabili nella grande distribuzione, tali architetture mettono a disposizione un numero di unità di calcolo normalmente compreso tra 2 e 10. I processori della seconda categoria dispongono comunemente di centinaia o migliaia di unità di calcolo, sono normalmente adibiti ad usi specifici ed hanno costi decisamente più proibitivi rispetto a quelli di un laptop.

Tuttavia, la mera duplicazione della capacità computazionale di un calcolatore non è sufficiente per aumentarne le prestazioni: architetture parallele richiedono programmi paralleli e i vecchi programmi sequenziali non sono più adatti per sfruttare le potenzialità offerte dai modelli *multi-core* e *many-core*, le applicazioni necessitano infatti di essere riscritte secondo appositi paradigmi di calcolo parallelo. Tali paradigmi vengono insegnati già da tempo all'interno dei corsi di laurea di informatica e ingegneria informatica: gli studenti imparano i fondamenti della programmazione parallela e hanno la possibilità di testare quanto studiato sui libri attraverso i loro PC equipag-

giati di processori *multi-core*. La situazione cambia nel momento in cui viene approfondito il funzionamento di architetture complesse come ad esempio i processori *many-core*.

La presente tesi rientra in questo contesto: essa si pone l'obiettivo di fornire agli studenti uno strumento efficiente per realizzare e testare applicazioni parallele per architetture *many-core*. Per questo motivo è stata indagata la tecnica dell'emulazione: essa consiste nella duplicazione delle funzionalità di un sistema *guest*, al fine di renderle disponibili in un altro sistema, chiamato *host*. Così facendo il sistema *guest* può essere testato utilizzando una sua versione software (un suo emulatore). Tuttavia, la realizzazione di un emulatore per architetture parallele *many-core*, consisterebbe in un considerevole numero di thread concorrenti, ciascuno dei quali emulerebbe una singola unità di calcolo; a prescindere dalla potenza di calcolo del sistema *host*, le prestazioni di un emulatore di questo tipo sarebbero troppo basse a causa dell'elevato numero di context switch tra i thread.

L'apporto di questa tesi consiste nel fornire supporto a questo tipo di elaborazione tramite l'utilizzo delle schede video, le quali offrono una notevole capacità computazionale normalmente adibita al rendering grafico. Parleremo infatti di *processori grafici*, o *GPU* (per *Graphics Processing Unit*), che sono generalmente composti da alcune centinaia di unità di calcolo denominate *core* e che da alcuni anni sono di fatto programmabili e di conseguenza utilizzabili per la comune elaborazione e non più esclusivamente per operazioni grafiche. Implementando all'interno di ciascun core della scheda grafica un thread, sarebbe possibile realizzare un emulatore per architetture *many-core* senza dover ricorrere ad hardware aggiuntivo (le GPU sono normalmente incluse nella maggior parte degli odierni PC o laptop).

Il problema è che le GPU offrono una grande potenza di calcolo attraverso processori *SIMD* (*Single Instruction, Multiple Data*), i quali eseguono la stessa istruzione in maniera sincrona utilizzando dati differenti; ciò pone un limite non da poco visto che l'applicazione parallela eseguita dall'emulatore potebbe prevedere flussi di esecuzione differenti. Per ovviare

a questo problema una nuova tecnica di emulazione è stata introdotta: essa descrive l'esecuzione di un algoritmo in termini di dati, i quali vengono ricevuti in input da un singolo programma che "adatta" la sua esecuzione in base al dato (all'algoritmo) ricevuto. Tuttavia il programma è sempre uguale, il che lo rende adatto ad essere eseguito sui core di una GPU.

È bene precisare che le librerie per la programmazione di GPU organizzano l'esecuzione dei thread sui core in modo da minimizzare la divergenza (ovvvero il tentativo da parte di più processi di eseguire differenti istruzioni all'interno dello stesso ciclo di clock); per questo motivo l'approccio classico di emulazione è stato implementato per essere confrontato con quello nuovo da noi sviluppato, al fine di delineare la possibilità e le modalità di un possibile approccio ibrido.

Contents

List of Figures

List of Tables

Part I

State of the Art

Chapter 1

Aims of this Thesis

Microprocessors based on a single processing unit dominated the market for more than two decades, increasing their clock frequency and reducing their die area [2]; this trend reached its limit around 2003 due to the high power consumption and the heat dissipation. Hence, processor manufacturers begin to switch their designs to models with more than a single computation unit (core), leading to the multi-core and the many-core architectures [12]. These two models differ in the number of cores, between two and ten for the former and several hundreds for the latter.

As the computational architecture moves from a single-core to a multiple-core model, sequential programs are no longer able to exploit the performance offered by processors. Indeed, they have to be explicitly rewritten in a multi threaded fashion and this require new specific programming paradigms.

Nowadays, common off-the-shelf PCs and laptops expose multi-core processors, making them an adequate testbed for parallel application developing (with the aid of high level libraries such as OpenMP [13] and MPI [15]). However, this is not true when considering the many-core architecture: writing applications specific for this kind of processor forces you to use hardware with high costs and that is usually not available in common markets. To overcome these limitations two different concepts are introduced:

Emulation Given a *guest* system S^G and a *host* system S^H, the *emulation*

consists in the implementation of the S^G functionalities on S^H. Using this approach, a parallel many-core architecture can be emulated using a laptop and parallel many-core applications can be executed (i.e., their execution can be emulated) on the laptop multi-core processor. However, this approach presents very low performances, due to the many-to-multi core mapping and the resulting high number of context switches.

Graphic Cards Graphics Devices (even those present in commodity PCs or laptops) are real many-core processing units, normally targeted to rendering, shading and texturing. They provide a big instruction throughput and a very high memory bandwidth. Thus, the Graphics Hardware is considered *Graphics Processing Units* (a.k.a. GPU). Starting from the early 2000s, GPUs have became programmable [2], allowing general purpose applications to be written using a high level language and executed on the top of a GPU.

These two concepts lead to the develop of a *GPU-based many-core architecture emulator*, such that real many-core applications can be written, tested and debugged on the emulator, while the required computational power is offloaded to the GPU cores. However, the graphics hardware does not direct support the execution of different flows. According to the Flynn taxonomy [22], the most suitable parallel model for our purpose should be the *MIMD* one (for *Multiple Instruction, Multiple Data*), this model allows multiple execution flows to be executed on different cores, working on different memory portions.

GPUs are many-core platforms that typically expose a *Single Instruction, Multiple Data*-like paradigm (*SIMD*), hundreds of cores are organized into several groups, each of which has a specific roles in the rendering pipeline. Cores from different groups are autonomous, but inside the same group they expose a *SIMD* behavior, where a single control unit fetches and broadcasts the same instruction to all processing elements [1], forcing them to

execute the same operation (*Single Instruction*) using different memory portions (*Multiple Data*).

GPU programming environment (like *CUDA* [21] and *OpenCL* [20]) usually implement a *Single Program, Multiple Data* (*SPMD*) programming paradigm, that is a paradigm for *MIMD* architectures. Each core runs an instance of the same program, using its own *Program Counter* following a specific path through the program [18].

The mapping from the *SPMD* paradigm exposed by GPU programming environments and the *SIMD*-like architecture exposed by GPU hardware is not trivial and it is not fully manageable by the programmer. Hence, some kind of task parallelism can fit GPUs, but some others cannot.

To investigate this topic, the classical emulation approach is compared with a new one. Typically, emulators work at the *Instruction Set Architecture* (*ISA*) level, providing a routine for each ISA instruction that is on-demand invoked every time the emulated processor fetches the corresponding opcode. Implementing such an emulator in each GPU core would cause different cores to emulate different execution paths, leading to the *divergence* phenomena (i.e., different instructions that should be executed in parallel during the same temporal step, are sequentially performed).

The new approach here proposed tries to answer the question "is it possible to express different computation using different data, handled by a unique algorithm?"

At a first sight, the answer could be a clearly "yes", the λ-calculus itself is a mathematical formalism for express computation where there is no distinction between programs and data.

But how this concept can be (efficiently) translate in practice? This thesis describes the emulation of a simple processor, where instructions are coded in terms of data that will be handled by a unique program. This approach become particularly interesting when considering GPUs as the *host* platform for the emulation. Now, each core can execute the same program, discarding all divergence issues.

For this purpose, processors are investigated at the *Micro Architecture* level, just a step above the rough hardware. Here, the hardware always performs the same execution path and different bits (the data) tune that execution to express different instructions behaviors.

Clearly, this approach has a cost: simple instructions that previously would have been emulated by few lines of code, now require one or more cycles that are executed by a software, and not directly by hardware. This thesis takes the classic approach (that is not well suitable for GPUs) and the new one (that has a high computational cost) and compares them, looking into the chance (and corresponding modalities) for a hybrid solution.

Chapter 2

Virtual Machines & Emulation

The majority of the computer science related topics are not concrete, the software is not a physical entity, but a virtual concept. However, within the computer science world the term *virtual* receives a specific meaning that express the ability to play the same role of another entity, offering the same interface to the outside world [16].

In computer science there are two main concepts: the *abstraction* and the *interface*, the former defines the operations an entity can perform, while the latter is the way by which these operations can be requested. For example, a software library defines new operations abstracting from the underlying levels (whether hardware or software) and it defines an *API* through which other entities can invoke the defined operations. The *virtualization* technique provides software entities that expose the same interface of another system, such that they can substitute it in every context.

This thesis investigates the virtualization concept through the *emulation* technique. Despite these two concepts are slightly different, they share the ability to implement a *target* system functionalities on a different one. The *Virtual Square* taxonomy [16] categorizes virtual machines according to their consistency to the lower layer interface. Virtual machines can be either *homogeneous* or *heterogeneous* depending on whether or not they provide the same interface of the system where they run. Processor virtualizers can oc-

cur in both modalities, on one hand *homogeneous* virtual machines allow the creation of a virtual environment with the same features of the system where the *VM* runs. On the other hand, a *heterogeneous* virtual processor permits a program compiled for a different architecture to be executed on the host system. Emulators are included in this latter category.

The *emulation* is a computer technique that duplicates the functionalities of a computer system (the *guest*) in another system (the *host*), this allows the host system to behave as a real instance of the guest system.

Let X be an either hardware of software entity, the entity *E(X)* (that can be either hardware or software too) is said to be an *emulator* of X if it allows another entity Y to be interfaced to *E(X)* as it would be interfaced to X.

Thus, a processor emulator allows to take a program compiled for an arbitrary architecture (e.g., i386, arm, mips, ppc, etc) and to observe its execution on the emulated processor, running on a different platform.

A video game console emulator is a program that run on either a computer or a video game console, it allows the execution of some games that were originally designed for a different console. For example, an emulator for a legacy video game console permits to use games for which the dedicate hardware is no longer sold.

"Emulation" is a black box term, that can be addressed by multiple points of view, this thesis focuses on the processor emulation, in particular on the many-core processor emulation. This kind of architecture exposes a high number (i.e., several hundreds) of processing elements, that differs from the multi-core architecture that nowadays offers between two and ten computational units, [2]. Following sections will explain some key concepts of the emulation technique.

2.1 Emulation vs Simulation

As a first step, it is important to distinguish between these two terms, that concern techniques with similar aims but with different effects.

The *emulation* technique allows the creation of entities that implement all functionalities of a guest (i.e., emulated) system. Emulators expose the same interface of the guest, so that other entities can interact with them in an unmodified manner.

On the other hand, the *simulation* technique produces environments that *mimics* the simulated system behavior. Hence, simulators expose interfaces that are different from the one exposed by real instances of the guest (simulated) system. Entities that can be interfaced with a specific system have to be rewritten when interfaced with a *simulator* of that system.

Consider for example a network simulator where packets are exchanged between simulated nodes occurring in some delays that are (hopefully) similar to real world network delays. In such a context, new protocols can be tested, observing how applications behave. But a network simulator is an unreal environment where nodes are not real ethernet-linked nodes, no packet is really exchanged and all delays are generated by the simulator. Real network applications cannot run on (i.e., they are not able to be interfaced with) the simulator. On the other hand, a network emulator forces applications to open sockets, to sends and to receive data packets as they would normally do [16].

Thus, the difference between emulation and simulation lies in the *interface*: while the simulation just mimics the target system behavior (it defines new interfaces, it requires new application), the emulation maintains interfaces unchanged, so that real unmodified applications can interact with the emulated system.

These two techniques have different use-cases, both of them present benefits and drawbacks and sometimes they compare coupled in a hybrid form. However, the emulation presents two main benefits:

- applications written (and tested) on the emulator can be executed on a real instance of the target processor, and vice versa (applications written for the target system can be tested using the emulator)
- specific hardware can be tested (i.e., applications for these architectures

can be developed) without really owning it

The latter point has a great importance in the education field, where students study from the books how to use specific hardware platforms but they cannot test them due to their high cost or their unavailability. In particular, this thesis deals with many-core processor emulation, giving students the chance to develop an application suited for a 100+ cores processor, using their own PCs or laptops.

2.2 Full system and User mode emulation

As said before, the term "Emulation" is a special case of the more general concept of *virtuality*, or *virtual entity* [16].

An entity X^V is a virtualization of X if it can efficiently replace (i.e., it can be used instead of) X itself. In this sense, emulation could be treated as a form of virtualization, it creates a virtual version of a either hardware or software component and this virtual component can be used instead of the real one (other entities can interact with it as they would do with the real one).

Thus, the term "emulation" no longer refers to a rigid *hardware-by-software* implementation, but *whatever entity* can be virtualized, in order to create its virtual version, that can replace the real one in every context. The expression "*whatever entity*" means that each system (from a small chip to a big and complex computer architecture) can be replicated by software via emulation/virtualization. This leads to multiple emulation approaches, depending on the complexity of the emulated system.

Firstly, a *full system emulation* duplicates the functionalities of a complete computer architecture, including instruction execution, memory management, I/O devices, etc. This mode permits to run a complete O.S. stack, testing all its functionalities.

Clearly, a full system emulator requires a considerable coding effort that sometimes could be useless: most of the emulated components could be al-

ready available in the *host* system. This leads to another approach, where only some parts of a complex system are emulated and coupled with the rest of the *host* system, which occurs in its real (not emulated) version.

This second modality includes the *user mode emulation* approach, which takes into account only the processor virtualization, discarding all other hardware components with which it is connected to. Hence, an unmodified program compiled for the *target* architecture can be "executed" on (i.e., its execution can be emulated by) a virtual processor.

Qemu is an example of a software that allows both presented modalities (see the next section for more details).

Pushing forward the concept of partial emulation (where only specific components are emulated), it is possible to emulate just a small part of a big entity. This thesis, for example, emulates only the execution path of the *Java Virtual Machine*. As discussed in chapter 1, the main goal of this work is the development of an emulator that can execute multiple execution flows in parallel, running on GPUs.

For this reason, the execution path will be the only focus of this thesis and components like interrupts handler and memory management unit are discarded. Furthermore, the choice of the *target* processor is not crucial (as discussed in chapter 4), the implemented ISA is a subset of Java Bytecode, that is much less powerful then the real Java language (no object oriented expressiveness, no input/output and integer only operations).

2.3 Available Virtual Machines and Emulators

As said in the beginning of the chapter, *virtual machines* and *emulators* are similar concept but they do not coincide; not all emulators are also virtual machines and vice versa. To clarify the differences between these two concepts, both virtual machines and emulators will be presented, highlighting the membership of each tools.

Qemu *Qemu* (short for *Quick EMUlator*, [17]) is a software that can be considered both an emulator and a virtual machine; it currently emulates different architectures such as i386, arm, mips and ppc. It allows the complete emulation of the target system in order to execute an unmodified O.S. in a virtual environment.

To achieve high performances, it implements a technique called *dynamic binary translation*, the first time a target instruction is reached *Qemu* translates it to a host system code fragment and stores it, so that it can be reused the next time the emulator reaches that instruction.

Due to its performance, *Qemu* can be used as a virtual machine instead of the real architecture in order to test, debug and run Operating Systems; moreover, it allows (only under Linux) the execution of programs compiled for a different architecture without having to start a complete OS stack (*User mode Emulation*).

According to the Virtual Square taxonomy [16] it can be considered a *Heterogeneous* virtual machine, since it expose to programs a different interface from the one exposed by the *host* system,

KVM, Virtual Box These two tools are virtual machines, but they are not emulators. Indeed they allow the complete virtualization of an architecture but they force the guest and the host systems to be the same. They are *Homogeneous* virtual machines.

To achieve near native performances, these tools make use of hardware-assisted virtualization, that permits a direct and fast instruction mapping from the target to the host (that clearly force these two architecture to be the same).

Java Virtual Machine As the name suggests, the *JVM* is a virtual machine that allows the Java Bytecode to be executed on an arbitrary architecture. However it is not considered an emulator in the strict sense; the guest system (the emulated one) is just an abstract specification and it does not appear in any real implementation, thus the *JVM*

is not considered an emulator since it implements the functionalities of a non-existing entity.

uMPS Finally, this is the case of an emulator that is not a virtual machine. μMPS is an educational computer architecture emulator developed at the University of Bologna, that implements the *MIPS I* Instruction Set [29]. Due to its educational goal it cannot be considered a virtual machine since it is not a good substitute for the real *MIPS* processor. It allows students to design an Operating System from scratch on the top of a simple hardware, controlling its execution step-by-step. However, using it in a real context implies unacceptable performances because the emulator goal is the correct execution of a program and not the time require for this execution.

Chapter 3

General Purpose Computing on GPU

The Graphic Processor (sometimes called *Graphics Processing Unit*, or *GPU*) is the engine element of the Graphic Hardware, the computer device responsible for the graphics elements management. Nowadays, GPUs are included in most of the off-the-shelf PCs or laptops, providing a dedicated hardware for the rendering process. To achieve high performances, this process makes use of several hardware components, each of which exposes a huge number of *synchronous* Processing Elements (*PEs*).

GPUs currently represent one of the most powerful computational hardware per dollar, moreover they expose a high memory bandwidth, making them an interesting device for non-graphic tasks too, through the processing elements exploitation, in order to achieve General-Purpose computation [4].

3.1 The Graphics Hardware

Graphics devices are today included in all commodity PCs, their task is the creation of a 2D image (i.e., a two dimensional array of pixel) starting from a scene description, provided in terms of an either 2D or 3D geometry, color, light and texture informations, [23, 24].

To achieve high performances, all GPUs manufacturers design their devices according to a well defined structure call *Graphics Pipeline* (or *Rendering Pipeline*). This pipeline includes five stages, each of which has a role in the rendering process that is performed by a dedicated hardware component. These steps are:

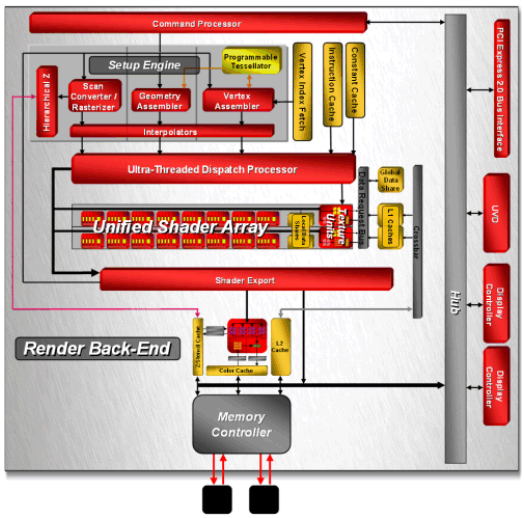
- Vertex Operations
- Primitive Assembly
- Rasterization
- Fragment Operations
- Texturing

Due to the parallel nature of the problem, *Rendering Pipeline* steps are usually executed by several *Single Instruction Multiple Data* (*SIMD*) processing elements working in parallel on multiple memory portions.

For example, Figure 3.1 shows the *Radeon RV710* structure [3] (the testbed GPU for this thesis); it exposes 4 Rasterizers, 16 Fragment Processor and 8 Texture Units. Although the *RV170* architecture is a low-end solution (it contains few processing elements), it is able to provide a big amounts of computational power.

In the following sections we will give an overview of the *GPGPU* concept, specifying capabilities and use-cases. As a preamble, it is important to note that one of its main aims is the ability to provide a high computational power without forcing developers to know low-level hardware details (though a minimal knowledge of the device behavior is clearly required). For this reason this thesis will not deal with the Graphics Pipeline process, leaving interested readers to [5].

Figure 3.1: A schematic vision of the *RV710* GPU architecture, taken from [3]



3.2 The *GPGPU*

The *General Purpose Computation on Graphics Processing Units*, from now *GPGPU*, is the exploitation of the Graphic Hardware (that is usually used for pixel management) for non-graphics tasks [6], this is a non trivial goal due to the Special-Purpose nature of this kind of devices. The next parts of this section discusses some GPGPU aspects.

3.2.1 An overview: when, what, why and how

In 1999 the *NVIDIA* company released the first programmable Graphics card: it can be consider the *GPGPU* birth. Before this date all rendering stages were hardwired; starting from the *Vertex* and the *Fragment* stages, all pipeline components were gradually transformed from fixed-function stages to developer-manageable programs. This capability, coupled with the introduction of an assembly language for stages programming, had enabled the *General-Purpose Computation for Graphics Hardware*, or *GPGPU*, term coined in the early 2000 by the *NVIDIA* itself. In 2002 another important result was achieved by the *ATI Radeon* company (now *AMD*) which has

introduced the floating point computation inside the *Fragments Operators*.

Beside these important results, a GPGPU community raised during these years [25], developing interesting solutions for the developers, some of which are described in the following sections.

Graphics devices offer some interesting features like big computational power, high memory bandwidth and a quick performance growth; despite this they are Special-Purpose devices. Thus, using them for General-Purpose computations presents some challenges. Firstly, the high computation power is offered through several *Single Instruction Multiple Data* (*SIMD*) elements, where a single control unit fetches and broadcasts the same instruction to multiple processing elements [1], that are forced to synchronously execute the same instruction using different memory portions.

While writing software for a single-core CPU is relative sample, the exploitation of a parallel architecture requires to write explicit parallel code; and since GPU processing elements are organized in a *SIMD*-like fashion, parallel code must be tuned for a *SIMD* architecture. Due to this fact GPGPU is recommended for computing intensive tasks while interactive programs are not well suitable for Graphics Hardware.

Moreover, GPUs provide high memory bandwidth for inner operations, like *load* and *store*, but the exchange of data between the Main Memory (the CPU) and the device memory is very expensive; so interactive programs are still not suitable for Graphic Hardware.

Fortunately, not all programs require an interaction with a human user or with some other device, there is a big class of GPU-suitable tasks like mathematical and physical computation (FFT [7], Matrix Multiplication) or graphics tasks itself (like Ray Tracing).

Beside these specific classes of tasks, there is another trend in the GPGPU that aims to treat the GPUs power in an even more general an flexible way, using it for any kind of computation, like it is a real coprocessor. Within this approach, it is possible to operate at different levels: [10] and [9] are example of GPU exploiting at the O.S. level, where GPU processing elements

are treated as any other O.S. managed resource over which processes can be scheduled. On the other hand, [8] and [11] use GPU computation power to speedup the virtualization and the simulation techniques, so that unmodified programs (compiled for a specific target platform) can be executed in a different context. This field has a special impact in this thesis, whose goal is the design of a many-core architecture emulator; however this is not a trivial task (as pointed out in [11]) due the SIMD-like architecture of the Graphics devices; this issue will be addressed in the second part of this thesis.

In recent years, GPGPU development tools have evolved providing high level solutions for GPU parallel applications developing; today these tools usually include a compiler from a C-like syntax to the GPU assembly language. Such assembly languages have evolved too, thus Graphics Instruction Sets does not only allow geometric primitives management, but some classical operations like mathematical, bitwise, memory and jump instructions are now included (memory and jumps operations will be deeper discussed in following sections).

3.2.2 Branches and *Divergence*

There is an important issues to address while studying *GPGPU*: since processing elements are clustered in a *SIMD* fashion, they are forced to synchronously execute the same instructions stream; but what happen if several *Processing Elements* evaluate the branch condition of an *if*-like statement to different boolean values? Conditional jump instructions have a key role in an assembly language since they are mandatory for loops description, in a full-fledged programming language conditions and loops must be available, otherwise the language capabilities would be very limited.

This issue can be addressed in several ways, the first one takes into account the GPU architectural design [23], branching can be direct implemented in hardware according to three modalities:

Predication This is not a strictly data-dependent branching, when the execution reaches a branch, both paths are evaluated and then, within

each *PE*, one path is discarded according to the boolean branch condition. It is the simplest method for branching support in GPUs and thus it exposes very low performances.

MIMD branching The *Multiple Instructions, Multiple Data* (*MIMD*) execution model would be the ideal hardware for branching support since each processing element is autonomous during the execution (i.e., it has a dedicated *fetch/decode* unit). To achieve this result, *MIMD branching* usually requires additional hardware components, that make the GPUs design more complicated. Apart from some *NVIDIA* cards, the *MIMD* branching support is not included in the majority of the GPUs [23].

SIMD branching This is the most widely adopted model: processing elements are organized in *SIMD* groups and when the execution encounters a branch instruction all boolean conditions are evaluated; if all these values are identical only one path will be executed. On the other hand, if one or more values differ from the others, both paths are executed and then each *PE* discards the result of the undesired path. (as in the Predication model)

The execution of both branch paths, discarding one of them according to the branch condition flag implies a great performance downgrade since a lot of cycles are wasted for a useless computation. The extreme case consists in multiple synchronously *Processing Elements* (*PEs*), each of which attempts to execute its own path; the available parallel power will be lost since all *PEs* would execute all possible branches, leading to a sequential-like execution. This phenomena is called *divergence*.

In addition to these, there are some further techniques to address the *branching* issue, which work at a higher level [26]; they try to establish if a branch path is useful or not, so its execution could be discarded in advance. This could be done either statically (with some *Branch resolvers*) or during the execution.

3.3 Available Tools

Any successful programming framework requires at least three additional components: a high level language, a debugger and a profiler [23]. Though profilers are very hardware specific (and their number is very limited), a lot of high level languages and debuggers are today available for GPU programming; this section focuses on popular languages, they can be organized in two main categories, *Shading languages* and *General Purpose languages*.

Languages from the first group share the common idea that the GPU main aim is the pictures creation, thus all the computation must be expressed in terms of graphics objects management. Languages belonging to this category are *Cg* (for *C for Graphics*, developed by *NVIDIA*), *HLSL* (for *High Level Shading Language*, a proprietary *Microsoft* language) and *GLSL* (for *OpenGL Shading Language*, from the *OpenGL Architecture Review Board* consortium).

Although it is quite simple to map these languages to a Graphics processor, they force developers to think a parallel application in terms of geometric primitives, vertices, fragments and textures, while General-Purpose algorithms are well described as memory and mathematical operators, that are concept much more familiar to classical CPU programmers.

For this reason, a second category of programming languages has been developed, in order to allow programmers to write GPGPU applications in a more familiar environment.

3.3.1 Brook

The *Brook* programming language is an ANSI C extension, developed at the Stanford University [27]. It allows applications to be designed in terms of *streams*, that are similar to *arrays* except from the fact that *streams* components can be accessed in parallel by a *kernel*.

A *kernel* is a routine that runs on the Graphics processor; due to the parallel nature of the GPU, multiple *kernel* instances will be executed using

the *stream* components as input.

In a *Brook* source, a *kernel* is a function, within its arguments it is possible to specify several input *streams* and one output *stream*. The source is precompiled by the *brcc* compiler, which transforms it in a C++ file, that can be in turn compiled using the standard *GNU C Compiler* tools. Within the so generated C++ file, the *kernel* function is transformed into a target specific assembly code that can be executed on various platform, including GPU and CPU itself.

3.3.2 CUDA and OpenCL

After *Brook* emerged, many similar solutions were born, in 2006 *NVIDIA* released the first *Computing Unified Device Architecture* (*CUDA*) SDK, a programming environment for parallel application developing. *CUDA* has quickly became the standard *de-facto* for *NVIDIA* GPUs programming.

In 2008, some industries defined the *Open Computing Language* (*OpenCL*) standard, that try to mimic the *CUDA* environment using a different aim: while *CUDA* works only with *NVIDIA* GPUs, *OpenCL* is a Heterogeneous Computing standard, it assumes multiple devices to be used for generic computations. Today, a lot of micro-processor manufactures provide a *OpenCL* implementation for their hardware. Despite the difference in the initial aim, these two programming tools are very similar, that is the reason why they are treated together. They expose a very similar programming and memory model, both described below.

OpenCL was developed in order to design a more flexible version of the *CUDA* environment, that does not force developers to use an *NVIDIA* device, but it allows the usage of any processing devices (like, for example, GPU, CPU and DSP). There is a strict correspondence between these two programming tools that allows to use terms from the two contexts interchangeably. Table 3.1 correlates terms from these two programming environments, moreover it shows a further column that contains other terms frequently used in literature for address the same concept.

Table 3.1: GPGPU programming nomenclature

OpenCL	**CUDA**	**Common literature**
Work-Item	Thread	execution elements
Work-Group	Thread Block	cluster
Compute Unit	Stream Multiprocessor	SIMD processor
Compute Device	-	GPU
Global/Constant memory	Global/Constant memory	-
Local memory	Shared memory	-
Private memory	Local memory	-

Finally, it is important to claim that even if they inherit the majority of the design from the *Brook* language, they cannot be considered *streaming* languages. On the contrary, both *CUDA* and *OpenCL* support the *Single Program Multiple Data* (*SPMD*) paradigm, where multiple processing elements execute different portions of a unique program, the *kernel*.

The programming model

The design of a GPGPU parallel application requires two parts: a *host* and a *kernel*, the former is a classical sequential application that runs on a common CPU with an Operating System like Linux, Windows or MacOS. A *kernel* is a routine that runs on the Graphic Processor, exploiting the device high parallelism.

One of the *host* job is the creation of the environment for the *kernel* launch, it consists in several phases:

(i) the establishment of the target device for the computation (there could be many devices in a system and the CPU itself could be used as a target device)

(ii) the compilation of the *kernel* code

(iii) the copy of data from the CPU memory to the GPU buffers

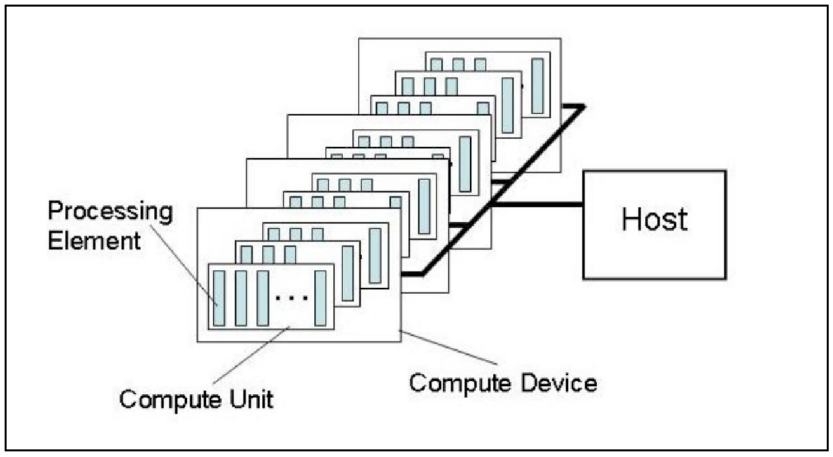
(iv) the choice of how many execution elements has to be parallel executed

After that, the parallel phase could start, at the end of which, the *host* application could copy back the results from the GPU buffers to the Main Memory.

To abstract from the different GPU architectures, *kernel* programmers could assume that the underlying hardware is organized according to a well-defined hierarchy; since each GPGPU library uses its own names to denote the same set of concepts, terms from the same line of Table 3.1 will be used in this thesis interchangeably, however since the main focus is on the *OpenCL* programming environment, terms from this framework will be mainly adopted.

The Figure 3.2 depicts the developer point of view, here the *host* application can interact with several *Compute Devices*, each of which consists of multiple *Compute Units* (*CUs*), within a *CU* the computation occurs through *Processing Elements* (*PEs*) that are clustered in some groups.

Figure 3.2: The OpenCL execution model, taken from [28]



A *work-item* (*WI*) is the software entity that logically correspond to an execution flow, the *OpenCL* runtime system maps each *WI* to a *PE* during the *kernel* execution and all *WIs* run the same code.

According to the input data, a *kernel* can organize its *WIs* in an N dimensional structure, with $N \in \{1, 2, 3\}$; *WIs* are clustered in group, called

work-groups (*WG*), each of which must contain the same number of *WIs*. OpenCL specification ensures that all *WIs* within a *WG* run in parallel on the same *CU*.

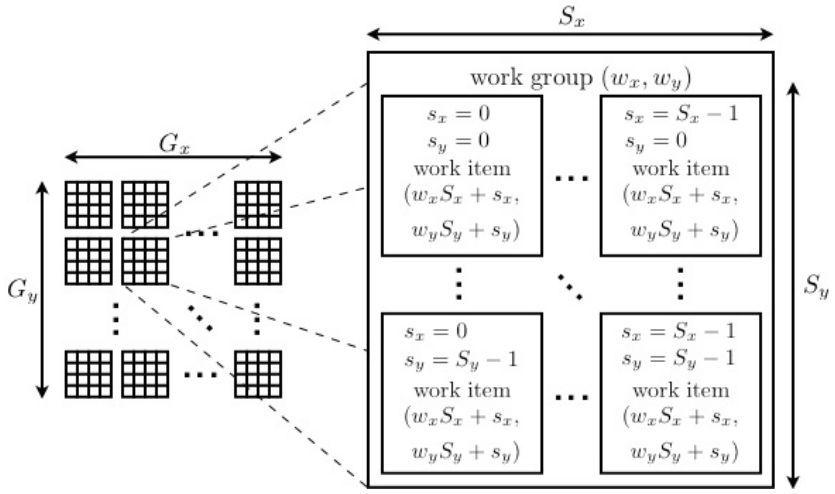
Once the *kernel* has been compiled, it must receive both the global number of *WIs* to execute and the size of a *WG*, the number of groups can be retrieved dividing these two values. Clearly, the *WG* size must divide the global number of *WIs*. As example, the Figure 3.3 shows a two dimensional *kernel* structure with $G_x \times G_y$ *WIs* clustered in groups with size $S_x \times S_y$. Each *work-item* can retrieve its position within both the *work-group* and the whole structure though the built-in functions

```
int get_local_id(int dimension);
int get_global_id(int dimension);
```

Both functions have an integer parameter that indicates the dimension over which the position must be retrieved. Figure 3.3 assumes that the numbering starts from the top left of the grid, counting from zero; the *WG* with index $(1, 1)$ is zoomed showing all *WIs* inside of it. Within a *WI* there are the s_x and s_y variables that hold the *WI* indices inside the *WG* (local indices). These indices could be retrieved using the first of the above functions (*get_local_id*), with arguments 0 and 1 respectively.

Once the *kernel* has launched, this model is mapped to the device architecture; this mapping could be more or less efficient depending of the device capabilities; since all *WIs* within a *WG* runs in parallel on the same *CU*, the maximum number of *WI* in a group is limited by the physical number of processing elements of a *CU* (regardless of the number of dimension). If an application requirement (i.e., the number of *WIs* to be parallel executed) is bigger then a *CU* capacity, multiple *WG* must be defined. It is possible to concurrently execute as many *WGs* as the number of the device *CUs*; *work-items* can coordinate themselves only within a *work-group*, thus if a device is equipped with multiple *CUs*, several *WGs* can be concurrently executed but *WIs* from different groups are not able to synchronize them.

Figure 3.3: A 2D Work-Group example, taken from [28]



The memory model

GPUs usually expose a high memory bandwidth such that various Processing Elements (vertices, fragments, rasterizers) can access the device memory with a high rate. However, this high performance is due to the parallel nature of the graphics operations, there is a strict correspondence between a processing element and the memory portion on which it operates.

When developing a General-Purpose application, programmers should be very careful since a wrong memory usage could cause a big performance downgrade, if all processing elements randomly access the whole memory area, a lot of time is wasted for the *PEs* synchronization in the bus usage. For this reason GPGPU programming environments usually expose a memory hierarchy, where different memory areas can be accessed with different capabilities.

As depicted in Figure 3.4, there are four different memory areas:

Global Memory It is the largest and the slowest area, both all *work-items* and the *host* can access it in both read and write modes.

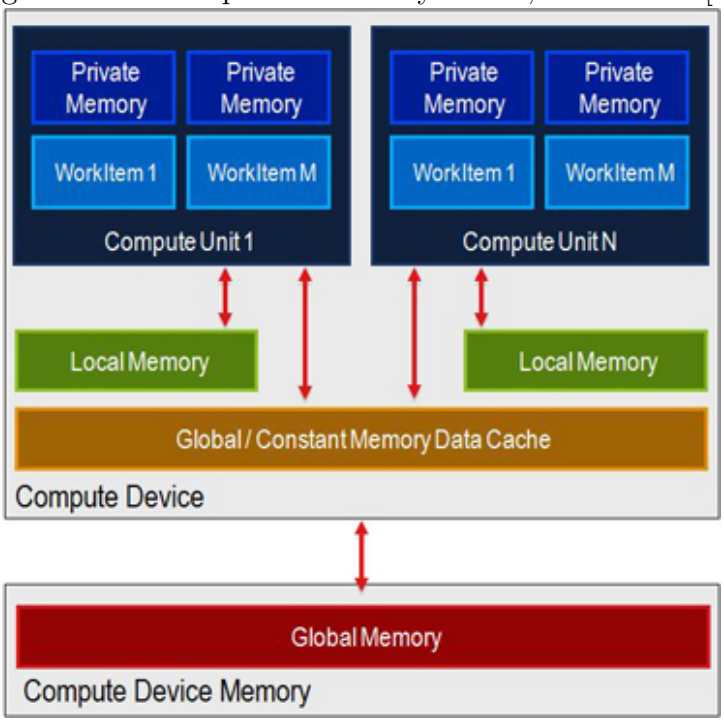
Constant Memory Like the *Global Memory*, this area can be accessed both by all *work-items* and by the *host*, the difference is that *WIs* cannot

modify its content. *Constant Memory* can be viewed as a read-only portion of the *Global Memory*. These two areas consist in the only communication channel between the *host* and the *kernel*.

Local Memory A memory region local to a single *work-group*, used for sharing variables between *WIs* belonging to the same *WG*. Since the limited number of processing elements competing for the bus usage, *Local Memory* has better performances w.r.t. both the *Global* and the *Constant* memory area. For this reason, it usually has a limited capacity.

Private Memory Data stored in this region is private to a single *work-item*, it is very fast and it is usually adopted for inner computations.

Figure 3.4: The OpenCL memory model, taken from [28]



The *CUDA* programming environment exposes a memory hierarchy very similar to the one described, except from the areas names, as show in Table 3.1.

The *OpenCL* programming environment uses a *relaxed* memory consistency model [28], this means that during a *kernel* execution, a memory consistent view from all *WIs* is not guaranteed. In particular,

- Private Memry is consistent within a *work-item*
- Local Memory is consistent across *WIs* inside a *WG*
- Global Memory is consistent across *WIs* inside a *WG*, but there are no guarantees of Global Memory consistency between different *WGs* executing a kernel

This model has a big impact for this thesis, a many-core architecture emulator should scale to a huge number of virtual processing elements, but this number is limited by the physical number of the available *PEs* of a GPU. To overcome this fact, multiple *work-groups* could be defined, in order to emulate a larger architecture. However, as pointed out above, between different *WGs* Global Memory consistency is not guaranteed, this issue will be addressed in the second part of this thesis.

Part II

A double level approach for Emulation

Chapter 4

The Emulated Architecture

"To emulate" means to provide an either software or hardware entity which expose the same functionalities of a *target* platform; this technique usually requires a good knowledge of the target system in order to allow other entities to be interfaced with the emulated system as they should do with the real one. For example, a processor emulator implements the target processor functionalities (the instruction set, the memory management unit, the interrupts handler, etc) in order to execute programs compiled for the target processor with no modifications.

However, the emulation does not always require the implementation of all components of the target system; taking into account only a small portion of a complex system permits to reduce the coding effort, exploiting the presence of components that do not have to be emulated.

That is the case of this thesis, it presents a *JVM* (*Java Virtual Machine*)-like emulator that limits its functionalities to the instruction set emulation, discarding the management of both memory and interrupts, since they do not concern this thesis goals. Indeed, this work investigates processors emulation in a parallel context using *GPUs* (*Graphics Processing Units*) to provide computational power support. Thus the choice of the target platform has not a big impact for the defined purpose, but whether parallel architecture can be implemented in order to evaluate the emulator performances.

We will refer to this platform as the *IJVM*, for *Integer Java Virtual Machine*; this name comes from the fact that the emulator mimics the JVM behavior but it limits its functionalities on those instructions operating only on integer values. Thus, the target language (the one that follows from this reduced set of instructions) loses all its object-oriented expressiveness because instructions working on object references are not implemented, making this Java-derived language a classic imperative language. The choice of the *IJVM* as the target processor comes from two main factors:

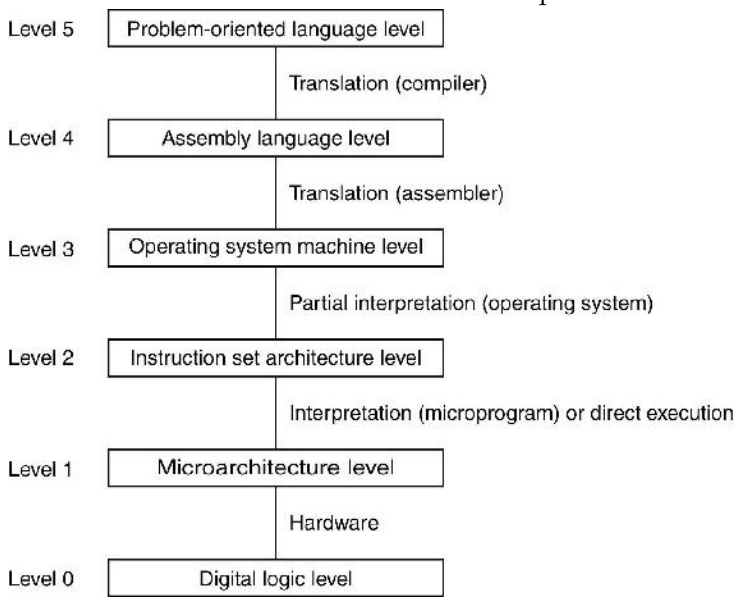
(i) Java is a high level language, that can be compiled to Bytecode, an Instruction Set whose instructions are easier to decode then those of a binary for a real hardware processor (e.g. i386/powerpc/arm/mips). This makes the Java language a good candidate for tests.

(ii) The processor execution path (in terms of hardware structure) is widely explained in the well known Tanenbaum book "Structured Computer Organization" [14]. Although the proposed model has not led to any real implementation, it offers a simple and accurate model for the execution path of the Java Bytecode.

In [14], Tanenbaum proposes a multilevel abstraction stack shown in Figure 4.1, where at the lowest position there is the rough hardware (Level 0, the *Digital Logic Level*) and growing towards the top, there are *Micro Architecture* and *Instruction Set Architecture* levels, respectively 1 and 2.

Usually, processor emulation techniques place themselves at level 2 (*ISA level*) of this model, implementing a specific routine for each instruction of the target processor. This thesis investigates a different approach, due to the fact that *host* platforms (the ones on which the emulated software runs) are GPUs. This detail has a great importance when the *target* architecture has a parallel design, indeed GPUs usually expose a lot of computation elements organized in a *SIMD* (*Single Instruction Multiple Data*)-like fashion, making the emulation of different execution flows not trivial.

Figure 4.1: The multilevel structure of Computer defined in [14]



If several SIMD cores attempt to execute different instructions during the same temporal step, these instructions are sequentially performed during distinct clock cycles (i.e., the *divergence* phenomena). Hence, GPUs computational power cannot be fully exploited using the classic approach.

This thesis proposes a new approach that moves one step down in the abstraction stack, taking into account the *Micro Architecture level*, the Level 1. Here instructions are no long treated as routines, but they describe different behaviors in terms of different memory words, which enable/disable different parts of the processor. This way, the emulation software no longer consists in a *switch*-like statement that invoke a specific routine depending on the fetched opcode (that is the common technique), but it iterates the following tasks:

(i) it selects an instruction

(ii) it splits the selected instruction into several fields (that are blocks of bits)

(iii) it composes these fields in a bit-wise fashion with the processor com-

ponents

(iv) it schedules the next instruction, restarting from point (i)

Thus, different computations (*task parallelism*) can be expressed in terms of different values stored in memory (*data parallelism*), making this approach suitable for GPUs usage.

Since GPU programming environments usually allow some form of *task parallelism*, the classic emulation technique is not completely discarded and both techniques are implemented and compared. The implementation details of both approaches are described in a more detailed form in chapters 5 and 6 respectively, the present chapter gives an overview of the target architecture, presenting its capabilities.

4.1 The *Integer Java Virtual Machine*

The *IJVM* is the architecture chosen as target for the emulation. As explained above, this choice has been influenced by two main reasons: Java Bytecode is both simple to write and easy to decode and its Micro Architecture level is fully described in [14]. However, this model has been partially modified in order to:

(i) be more efficient

(ii) support parallel execution

(iii) support array dynamic allocation

Except from these factors, the emulated architecture is the one described in [14], it uses an *Instruction Set Architecture* (*ISA*) that limits its possible instructions on those working on integer values and integer arrays.

In a parallel scenario, there are two possible memory models: the former is the *Shared Memory* paradigm, it expose a unique memory address space and parallel processes communicate reading and writing shared variables.

The latter is the *Distributed Memory* model, each process has its own memory address space and the communication occurs via message passing, [18]. Clearly, these are general models, real world applications actually implements a hybrid form of these two paradigms.

The *IJVM* version emulated in this thesis implements a form of the *Shared Memory* paradigm. Since the implemented language is Java, parallel applications are designed from classes. The shared memory is realized using class *fields*, while processes are defined using *methods*. Each method has its own local variables that are not visible by other methods. Thus, each process will have a dedicated *Local Variable* area, a dedicated *Stack* area (for local computation) and a *Global Variable* are, shared with other processes.

The parallel execution mimics the *Fork/Join* model, several parallel processes are dynamically created (*forked*) during the execution and there is a point within the program where the execution stops until each forked process reaches that point (the *joining* phase) [18]. The *IJVM* execution starts from the *constructor* method and each GPU core executes the code of the constructor. Every time a method invocation occurs, only one core *forks* and begins the called method execution, while other cores continue with the constructor code, waiting for further methods invocation. The *joining* phase is implicit with the method termination, thus when a process reaches its *return* opcode, the control does not return to the constructor but the *return* statement is repeated until all processors reach their corresponding *return* instruction.

Clearly, when the number of method invocation reaches the number of available core, no more method will be called.

Object management is not allowed, except from array (that are treated as object in Java). For this reason, there is a further memory area, called *Heap* for dynamic arrays allocation.

4.1.1 Registers

In the *IJVM* model all the computation takes place within the *Stack*, ALU operands are pushed into the stack and replaced by the ALU result. To support this kink of computation, there some Special-Purpose registers:

- MAR, MDR, PC, MBR: for *memory operations*. *Micro Architecture* level provide two different memory access modes and these four registers behave respectively as source and destination for these modes. At the *ISA* level these register will not be considered since memory operations are explicit. On the contrary, they have a key role at the *Micro Architecture* level

- SP: the *Stack Pointer*, it is the address of the last value pushed into the Stack

- LV, CPP: *Local Variable* and *Constant Pool Portion*, these registers store pointers to the beginning of the LV and CPP areas in the memory respectively (see next section for more details)

- TOS: *Top Of Stack*, the value stored in the memory position pointed to by the SP register

- OPC: *Old Program Counter*, when invoking a new method this register will hold the value of the caller PC

- H: *Holding*, the only general-purpose register. It is usually used as intermediate storage for complex computations

4.1.2 The memory structure

In [14], the proposed model exposes a 4 GB large memory, organized in 32 bits words.

There are four areas:

CPP The *Constant Pool Portion* is a read-only byte-oriented area that contains some informations about methods (the starting address, the number of arguments) and fields

LV The *Local Variable* area stores variables local to a method, indexed according to the order by which they are declared in the source

Stack It is the memory area where the computation takes place

Text It contains the program: opcodes and operands are stored in this area. Like *CPP*, it is read-only and bye-oriented

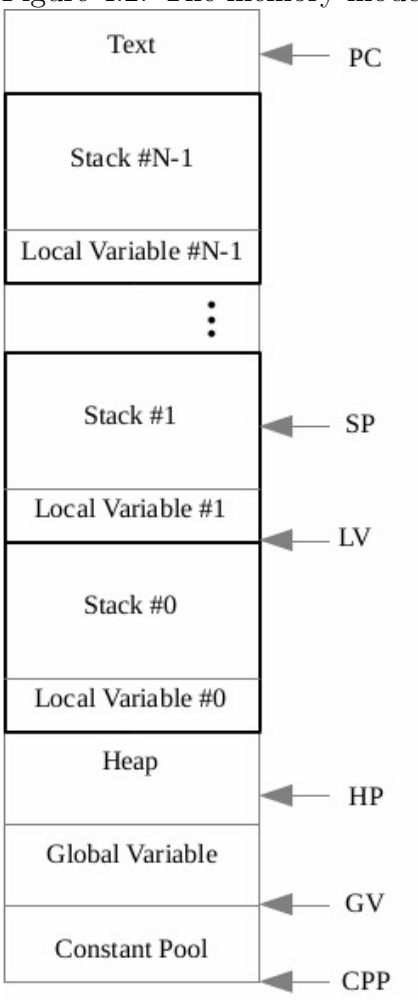
This model has been partially modified according to the parallel execution support and the dynamic array allocation. The *LV* and the *Stack* areas are replicated for each process. Furthermore, a *Global Variable* area and a *Heap* area are added, in order to allow a shared memory communication channel between processes and a way for the dynamic array allocation respectively.

The dynamic memory allocation is very coarse, every time a `new` instruction is fetched, the current *Heap Pointer* is increased by the required size and the older value is returned. Thus, there is no *memory deallocation* and no *garbage collection*.

This refined memory model requires the introduction of the *Heap Pointer* (*HP*) and the *Global Variable* (*GV*) registers; they will act as pointers to the new memory areas (like *CPP* and *LV* point to the base of *Constant Pool Portion* and *Local Variable* areas respectively). In addition, sine ither registers are introduced within the model in order to simplify the computation: the *Stream Identifier* (*SID*) register, for the identification of the current process, and the *Invocation Counter* (*IC*) register, that stores the number of invoked methods.

The implemented memory architecture is depicted in Figure 4.2, the right side of the picture shows some registers pointing to specific memory areas, while the *CPP*, the *GV* and the *HP* registers are common to all processors, each core has its own *SP*, *LV* and *PC* registers that define the computation status.

Figure 4.2: The memory model



The addressing occurs via 32-bit words, thus the memory could be theoretically 4 GB large. However, the GPU must store some further information in addition to the memory array and, as explained in chapter 3, graphic devices usually expose a memory hierarchy where different memory areas can be accessed by different cores with different performances. The higher the number of core sharing a memory area is, the higher the resulting access time will be, GPUs expose both little and fast private memory areas and large but slow global memory areas.

Hence, the *IJVM* memory capacity must deal with the GPU memory architecture, that can varies between devices.

4.2 The emulation of the *IJVM* model

Previous sections have described the *IJVM* model, highlighting the fact that *host* systems are GPUs. This fact creates some restrictions in the develop of a many-core processor emulator. Since GPU cores usually implement a *SIMD*-like paradigm, the classic emulation approach that invokes a specific routine for each instruction is not well suitable. The parallel emulation of different instruction flows would cause GPU cores to diverge, losing parallel

power (see chapter 3 for more details).

GPGPU programming environments (like Brook, CUDA or OpenCL) expose a *SPMD* (*Single Program Multiple Data*) paradigm [2], that makes each GPU core to execute the same program using its own *Program Counter*, thus each execution is independent from the others. However, this programming paradigm has to deal with the *SIMD*-like organization of the hardware. GPU cores are clustered into some groups, cores within the same group execute in a pure SIMD manner, but cores that belong to different groups can be considered autonomous. This "cluster organization" can be partially managed by programmer, but is always under the control of the adopted library, which arranges the execution flows according to some internal rules.

As a consequence, GPUs expose a massively parallel architecture that can partially support a *MIMD*-like execution. This thesis investigates how this *MIMD*-like execution can be exploited, comparing two different emulation approaches: the former emulates at the ISA level, implementing a specific routine for each ISA instruction (that is the classic approach). The latter moves one step lower in the abstraction stack (see Figure 4.1), implementing the Micro Architecture level where the computation is no longer expressed in terms of routine, but different memory words enable/disable specific components of the (emulated) processor. At this level, the emulation consists in the implementation of the processor components (registers selection, ALU computation, memory operations, ...) and the execution is always the same, regardless of which instruction has to be performed. Unlike ISA level emulation, this approach is suitable for GPU hardware, because SIMD-arranged cores can emulate the execution of different tasks implementing the same code.

Both approaches are implemented and compared, the latter (the Micro Architecture one) is improved exploiting the parallel computation power exposed by GPUs, leading to three distinct models:

ISA the classic emulation mode: every ISA instruction is implemented by an ad-hoc routine. Although it could be the more intuitive approach,

it presents a non trivial scalability issue: the more tasks are parallel emulated, the more they will diverge, leading to a sequential-like execution

MIC A precise emulation of the Micro Architecture model described in [14]: each task is emulated by the execution of the same code, making the ISA instructions implementation effort much more expensive, with a corresponding performance downgrade

PMIC (that stands for *Parallel MIC*) it is an improved version of the *MIC* model. This refined architecture provides three distinct processing elements for each instruction flow (that can perform up to three distinct operations in parallel)

These models are compared, in order to discover if and how GPUs are good device for an the emulation hosting; ISA emulation could obtain higher performance in terms of execution time, but it scales worse than MIC approach.

Starting from these observations, it could be possible to develop a hybrid solution that mixes the GPU-suitability of the MIC model and the flexibility of the ISA model.

4.2.1 Evaluation

The two proposed approaches are compared and evaluated according to some measurements. The first one considers the number of processes concurrently emulated as the variable factor and two different performance measures: the average execution time per instruction and the global memory usage. Given N The former measures the time required for the execution of a single instruction. If an emulation run requires t seconds to be completed and the N processes are composed of $n_0, n_1, .., n_{N-1}$ instructions respectively, the execution time per instruction $T(N)$ is:

$$T(N) = \frac{t}{\sum_{i=0}^{N-1} n_i} \tag{4.1}$$

The global execution time measurement depends on both the number of parallel processes and the amount of processes instructions. Since the latter factor does not affect the scalability of an emulation approach the execution time has been normalized according to the number of processes instructions, obtaining the execution time per instruction measurement.

The second evaluation is the global memory usage, it consists in the memory capacity required by an emulator, i.e. the Mega Bytes that have to be copied to the GPU buffers. Different approaches have different memory requirements, typically Micro Architecture level emulation require more memory than the ISA level emulation, sinc the code has been translated into data. Hence, an execution time improvements can corresponds to a memory occupation increase, that sometimes could be intractable (as explained above).

Chapter 5

ISA level emulation

This chapter presents the classic processor emulation approach, that permits an unmodified program compiled for a specific architecture to be executed on a different platform.

As discussed in chapter 2, "to emulate" means to provide an either hardware or software system that exposes the same functionalities of the emulated system. The more intuitive approach for the development of a processor emulator considers the processor instructions as the primary entities to be emulated, for this reason this approach will be referred to as the *Instruction Set Architecture* (*ISA*) level emulation.

At this level the emulator provides an ad-hoc routine for each opcode belonging to the *Instruction Set* of the emulated processor, the processor image is described through specific data structures for the Registers block, the Main Memory and the Interrupt Vector.

5.1 The *Fetch-Decode-Execute* cycle

Given a target architecture (e.g. *i386*, *arm*, *mips*, ..) an emulator takes a program compiled for that platform as input and it mimics the *Fetch-Decode-Execute* (*FDE*) cycle in this way:

Fetch it fetches a byte from the binary file

Decode it selects the right routine through a *switch*-like statement according to the fetched opcode

Execute it invokes the chosen routine, providing the processor image as input

If the adopted language permits the use of function pointers, the *Decode* and the *Execute* steps can be merged in a unique phase. Opcode routines can be referenced by a function pointers array `opcode[]`, where the generic element `opcode[i]` points to the routine corresponding the opcode *i*.

These steps are repeated until there are no more opcodes to be fetched or a *return*-like statement is reached.

5.2 A parallel architecture emulator

The aim of this thesis is the development of a many-core architecture emulator such that multiple binary files can be taken as input and concurrently emulated. Thus, there will be several virtual *Execution Units* each of which emulates its own *FDE* loop.

In a single-core scenario, these *Execution Units* are treated as concurrent processes (or threads) that are interleaved on the processor core. Due to the high number of context switches, this approach leads to a big performance downgrade, that can be reduced using an either multi-core or many-core platform as the *host* system for the emulation. The emulator performs a many-to-many mapping from the virtual *Execution Units* to the real processor cores.

This thesis investigates this approach, it implements a many-core emulator on the top of a *Graphics Processor*, the mapping is very simple since once an *Execution Unit* is scheduled on a GPU core, it performs all its computation on that core, reducing the mapping time.

As discussed in previous chapters, GPU cores are arranged in *SIMD* groups, within which they are forced to simultaneously execute the same

instructions sequence using different memory portions. Thus, the parallel emulation of multiple execution flows is not trivial, indeed given N input programs to emulate, for each of them there will be a dedicated *Execution Unit*, which iterates the *Fetch*, the *Decode* and the *Execute* phases. Though the first two steps (*Fetch* and *Decode*) could be easily emulated on N *SIMD* cores (all cores would perform the same instructions sequence for these steps), the *Execute* phase always requires different instructions to be performed, leading to a divergent execution.

To address this issue, this thesis proposes a new emulation technique, that will be detailed described in the next chapter; this new approach consists in a unique routine R where each opcode is treated as a processor configuration that allow R to behave differently depending on the opcode. It is important to claim that *GPGPU* libraries usually apply some policies when scheduling parallel threads on GPU cores, in order to reduce the divergent phenomena as much as possible.

For this reason, both emulation techniques are implemented and compared.

5.2.1 Improvements for the *ISA* emulation

The *ISA* level emulation is based on a mature technology, with a lot of innovations proposed during recent years (see chapter 2). For example, the *Qemu* machine emulator implements a combination of static compilation and dynamic translation to achieve high performance [17].

However, this technique cannot be direct implemented on the top of a *GPU* since this kind of device does not allow all programming techniques that are commonly available in the classic CPU programming (e.g., function pointers); moreover some new techniques need to be introduced in order to address the *divergence* phenomena; in [11], a many-core emulator is developed on the top of a GPU, introducing a further compilation step where the input program is compiled to a new *ISA* with a very limited number of instructions, in order to reduce the divergence probability.

The *ISA* level emulator developed for this thesis adopted a similar technique, instead of writing a dedicated routine for each opcode, instructions are grouped by type and each type corresponds to a single parametric routine, that is invoked with correct parameters every time an opcode of the corresponding type is fetched. For example, the `add` and the `sub` instructions have the same behavior except from the sign of the second operand:

$$a - b \equiv a + (-b) \tag{5.1}$$

Thus, there will be only a single routine, e.g.

```
void add(cpu_conf_t *conf, int sign);
```

that will be invoked with $sign = 1$ for the `add` instruction and with $sign = -1$ for the `sub` instruction.

As a more significant example, consider that within the *Integer Java Bytecode Instruction Set* there are 12 *jump* instructions, 7 *constant* instructions, 8 *load* instructions and 8 *store* instructions. For each of these groups there will be a unique routine.

Chapter 6

Micro Architecture level emulation

This chapter wants to answer the question "*is it possible to express different computation flows, using a unique algorithm?*". Let P be a problem, an algorithm A solves P if starting from an input dataset (even empty) it produces a solution for P (the output).

The goal of this thesis is the design of an algorithm able to mimics the behavior of a generic algorithm A, taking both the description of A and the dataset on which A operates as input. Hence, the steps for solving P are no longer expressed as "*algorithm steps*" but as part of the "*input dataset*".

At low levels computers act in a very similar manner, programs consist of instructions streams that are stored inside memory as binary files. The hardware always performs the same execution path and instructions opcodes are just numeric values that represent a specific hardware configuration. The hardware can "behave" in different ways, according to different configurations, i.e. different instructions.

This is the idea exploited in this thesis, the book "Structured Computer Organization" by A. S. Tanenbaum [14] describes in a very detailed manner the Micro Architecture level of a processor able to execute simple Java programs. Here, the *micro-instruction* is the basic computation unit, it

is composed of several bits, each of which manages a specific part of the hardware configuration, an instruction is an index within an array of *micro-instructions*.

6.1 The execution model

The model described in [14] does not correspond to a real processor description. Since the book has an educational purpose, it takes the Java Virtual Machine just as a case-of-study to explain processors design rules and technologies.

As a consequence, the proposed architecture is quite simple to understand but it has very low performances.

However, the simple and easy-to-understand example (called *Mic-1*) is made more complicated and more efficient in several ways, leading to other three abstract models, called *Mic-2*, *Mic-3* and *Mic-4*. These improved models highlight some key points in the processors design and they are studied as possible features for our GPU-based parallel emulator.

In the *Mic-1*, the primary entity is the *micro-instruction*, a set of bits that describe how various processor components should behave. These components are depicted in Figure 6.1:

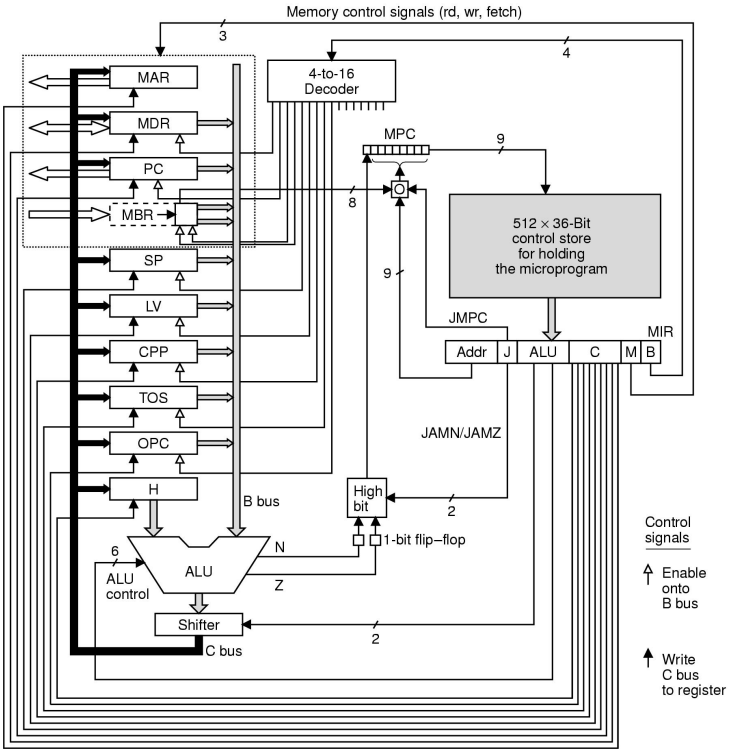
Control Memory a Read-Only memory that contain the *Micro Program*, i.e. the micro-instruction set. The term "program" is quite misleading because it is not a program in the strict sense, but a set of micro-instructions. However, the term will be kept as it is presented in the book. Within the Control Memory micro-instructions are usually hardware coded

ALU an Arithmetic-Logic Unit. It is able to perform some basic operation like sum, difference, increment and logical conjunction, disjunction and negation. It receives two values from two input buses and it sends another value on an output bus. Both input and output buses are connected to the registers

SP Registers Special-Purpose registers, they have specific roles in the computation. They are the only input/output for the ALU, thus the data stored in the Main Memory has to be copied into some register (with a *Read* operation) and then, the ALU can use it. See section 4.1.1 for more details.

Main Memory memory area for code, methods, variables and stacks; this component is not showed in Figure 6.1), for more details see sections 4.1.2 and 6.1.3

Figure 6.1: The Mic-1 execution Path, taken from [14]



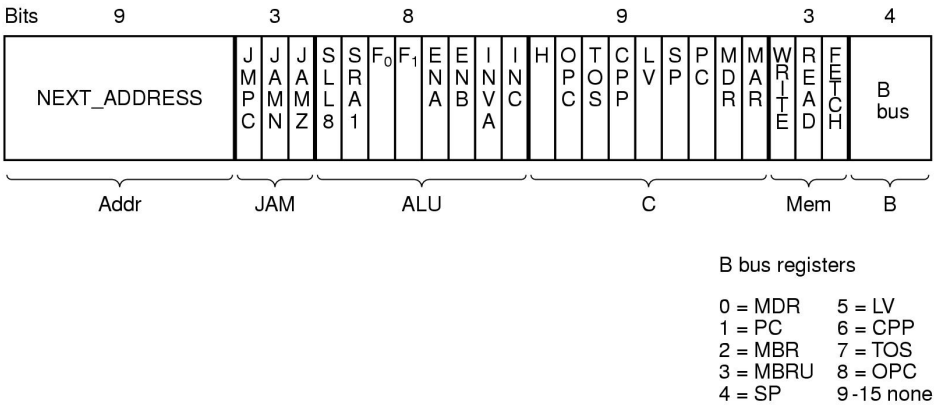
In addition to those listed in section 4.1.1, there is a specific register, called *MicroProgram Counter* (*MPC*) that acts as an index inside the *Control Memory*, it selects the *micro-instruction* whose signals are broadcast to the *ALU*, the registers and the Main Memory, These signals define the hardware configuration that produce the desired computation.

6.1.1 The micro-instruction

If the ISA level is instruction-oriented, at Micro Architecture level the primary entity is the *micro-instruction*. This entity is nothing more that a set of bits, grouped in fields that express how processor components should behave, in order to perform the desired computation.

In particular, the *Mic-1* micro-instruction is a 36 bits long word with 6 fields, as shown in Figure 6.2:

Figure 6.2: The Micro-instruction structure, taken from [14]



B Bus 4 bits that express which register (only one at a time) can put its value on the right bus of the ALU (the *IJMV* model does not give the chance to control the left bus, called A, that always receive data from one specific register)

Mem 3 bits that denotes which memory operation the micro-instruction should perform. The available operations are *Fetch*, *Read* and *Write*; differences between these operations will become clear in section 6.1.3.

C Bus 9 bits that define which registers will store the computed value. Unlike the input bus (the B Bus) that can receive a value only from one register, this output bus can bring data to multiple registers; for this reason this field needs 9 bits instead of *B Bus* field that needs only 4 bits

Alu 8 bits that control the ALU behavior, the first six bits refer to the ALU itself, while the last two to the Shifter

Jam this field is close related to the subsequent *Next Address* field, that specifies the index of the micro-instruction to be performed after the current one. The bits from the *Jam* field allow to modify this "next micro-instruction" field: first two bits invert the more significant bit of the *Next Address* field according to a *Null* or a *Negative* result of the Alu respectively. The activation of the third bit forces to use the value stored in a specific register (the PC register) as index for the next micro-instruction

Next Address these 9 bits act as index for the subsequent micro-instruction. At the Micro Architecture level, micro-instructions are a well defined number of building blocks that can be composed together in order to perform a more complex computation. They are usually hardware-coded and stored in the *Control Memory* (see Section 6.1); so the order by which they are stored usually is not the order by which they are executed. For this reason, the flow of instruction has to be explicitly coded in this field.

6.1.2 The ALU

The *ALU* (for *Arithmetic Logic Unit*) is the processor component that performs mathematical and logic operation. In addition, it is responsible for some decision operation (e.g., conditional jumps depends on the ALU result).

It is connected with several Special-Purpose registers, which have specific roles in the computation, as explained in section 4.1.1. The *ALU* has two input buses and one output bus, each of which is connected to registers.

It is important to note that only one input bus (the right one, called *B*) can receive a value from an arbitrary register, while the other one (the left one, called *A*) always receives data from one specific register (called *H*, for *holding*). Furthermore, not all registers can neither provide values for

the input bus nor receive values from the output bus: the micro-instruction provides two specific fields (*B Bus* and *C Bus*) that specify which register/registers has/have to be considered.

6.1.3 The memory model

As described in section 4.1.2, the Main Memory has four areas: *Constant Pool Portion* (*CPP*), *Local Variable* (*LV*), *Stack* (*Stack*) and *Method Area* (*Text*). These areas store informations about methods, variables local to a specific method, the stack on which computation takes place and the methods' opcodes, respectively. This organization is similar to the *Linux-IA32* model with .TEXT, .DATA, .STACK and .HEAP segments.

Registers are the only input/output for the ALU. Hence, when input data is not available in registers (it is stored in the Main Memory), there are three distinct steps to be performed:

(i) read the operands from the Main Memory to some registers (one at a time)

(ii) perform the desired operation

(iii) store the result back to the Main Memory

Te data exchange between the Main Memory and the registers takes place with two different modalities: *word-oriented* and *byte-oriented*; the former permits to read and write 32 bits words from the memory to the registers and vice versa. The latter allows only to read one byte from the memory, storing it in a register.

The presence of two distinct memory access modes is due to the fact that data stored inside the Main Memory can be used in two different ways: since the ALU performs operations on 32-bit words, operands for computations require a *word-oriented* mode. However, the Java Bytecode stored in the *Text* area and the methods informations stored in the *CPP* area are expressed in

byte. Hence, methods informations and opcodes require *byte-oriented* access mode.

Thus, there are three distinct memory operations:

Fetch It extracts one byte from the memory location pointed to by the PC register, storing it in the less significant 8 bits of the MBR register

Read It copies a 32 bits word from the memory location pointed to by the MAR register to the MDR register

Write It copies a 32 bits word from the MDR register to the memory location pointed to by the MAR register

Furthermore, the *fetch* operation has two versions: *unsigned* and *signed*. The first one treats the retrieved byte as an unsigned, 8-bits long, integer, with value range [0, 255]. The second version uses the more significant bit as the sign, leaving other 7 bits for the value itself, with a possible value range of [-128, 127].

Registers names reflect their usage: MAR stands for *Memory Address Register*, MDR for *Memory Data Register*, MBR for *Memory Byte Register* and PC for *Program Counter*, which holds the index of the current opcode within the *Text Area* of the Main Memory.

An important aspect of the memory management is the timing, the model described in [14] mimics a real hardware implementation, hence the author discusses all signals propagation related issues. For example, to perform a read operation the MAR register has to be set with the appropriate address (the Main Memory location to be read) and the micro-instruction must have turning on the specific bit in the *Mem* field on. If this happen at the cycle i, the value inside the memory is read during the cycle $i+1$, and it is available inside the MDR register from the cycle $i+2$.

Fetch and *Write* operations behave exactly in the same way.

However, these hardware details could be ignored when developing an emulator. Since the data coming from the Main Memory and from the registers is available with the same delays, it is possible to complete a memory

operation in the same cycle when it has been launched. In this way, the number of emulated cycles should reduce, because each memory operation requires one less cycle to be completed.

6.1.4 The execution path

The *Execution Path*, is the operations sequence that permits the *Mic-1* model to perform a computation. At the Micro Architecture level, "to execute a program" consists in:

(i) select from the *Control Memory* the micro-instruction pointed to by the MPC register

(ii) propagate bits from the micro-instruction fields to various processor components, in particular:

- *Bus B* and *Bus C* fields select input and output registers for the ALU, masking other registers
- *Alu* bits define the ALU operation
- *Mem* field bits enable/disable memory exchange between memory and registers, as specified in section 4.1.2
- *Jam* and *Next Address* fields establish the following micro-instruction to be performed (i.e., they store the following micro-instruction index inside the MPC register), according to the ALU outcome and the value stored in the PC register

(iii) restart this cycle from its beginning, executing the next micro-instruction which is the one pointed to by the newly value of the MPC register

6.1.5 An example

At this level, micro-instructions are coupled in order to provide ISA instructions implementation. For this purpose, micro-instructions are expressed with a high-level notation to improve readability.

This notation defines the ALU operation (with corresponding input and output registers), the possible memory operation(s) and the next micro-instruction to be performed. Here there are some examples:

```
main1: pc=pc+1; fetch; goto(MBR)

iadd1: mar=sp=sp-1 ; read ; goto iadd2
iadd2: h=tos ; ; goto iadd3
iadd3: mdr=tos=mdr+h; write ; goto main1

swap1: mar=sp=sp-1 ; read ; goto swap2
swap2: mar=sp ; ; goto swap3
swap3: h=mdr ; write ; goto swap4
swap4: mdr=tos ; ; goto swap5
swap5: mar=sp-1; write ; goto swap6
swap6: tos=h ; ; goto main1
```

The adopted notation provides several lines, each starting with the corresponding micro-instruction name, after that there are three parts, separated by a semi-colon. The first one, *the ALU computation*, defines the operation and the involved registers, not all registers are good candidates for either the L-value or the R-value of the computation, indeed, some registers are ready-only and some others are write-only. Regarding the ALU operations, there is a strict set of available operations.

The second part refers to *the memory operations* (fetch, read or write): read and write are mutual exclusive, but each of them can be performed together with fetch.

The last part specifies the name of the next *micro-instruction* to execute.

In the example, the Java instructions *iadd* and *swap* are implemented with several micro-instructions, that are executed with the ordering in which they are wrote (each micro-instruction "call" the next one). Both micro-instruction sequence end by calling the *main1* micro-instruction. Although *main1* does not refer to any instruction from the Java Bytecode ISA, its

presence is required: its only task is the increment of the *Program Counter* register and the fetching of a new opcode, in order to start a new operation.

Since a *fetch* operation copies one byte from the memory location pointed to by the *PC* register to the less significant part of the MBR register, the *main1* micro-instruction ends with the *goto(MBR)* statement.

6.2 Improved models

The *Mic-1* is an easy-to-understand example of Micro Architecture design, it is able to execute sample Java programs. However due to its educational purpose, it is not very efficient: the absence of a pipeline, of a branch predictor, of a cache memory and a long execution path make a Micro Architecture design like this not even comparable to modern CPU design (like for example the *Intel Core* Micro Architecture).

Tanenbaum book starts from the *Mic-1* model and improves it, leading to other three models, called *Mic-2*, *Mic-3* and *Mic-4* respectively. Now it follows a briefly description of the novelties introduced by these models, discarding a precise description of the models themselves and the benefits introduced w.r.t. previous versions. Instead, these improvements are discussed taking into account that the emulator *host* platforms are GPUs since some of these techniques can be either not suitable or hard-to-implement in a GPU application.

Prefetching Some instructions require a defined number of arguments (or operands) and *Mic-1* model must perform a fetch operation for each of them, increasing the number of cycle and, consequently, the global time. The *Mic-2* model introduces a sort of prefetching, that treats the memory as a byte stream. Every time a byte is fetched, the subsequent byte is fetched too, thus, if a micro-instruction requires an operand, it is already available. Moreover, the *Mic-2* model introduces a new register *MBR2*, that has a similar role to MBR except from the fact that

it holds two bytes instead of one. It is typically used for instructions that need more that one operands.

Applying this concept to a GPU-based emulator is not too difficult: the emulator proposed in this thesis exploits the dual memory access mode: *word-oriented* and *byte-oriented*. Each operation is treated as a word-oriented one and when a *fetch* operation is requested, four bytes are fetched (instead of one) splitting them up in four different registers (MBR0 to MBR3). Thus, all memory operations are similar (they can be performed in parallel) and when an opcode is fetched, the following three bytes (possible operands) are fetched too.

Pipeline and Branch Predictor The pipeline has a leading role in nowadays CPU performance: the basic idea is to divide a process (in this case the execution path) in multiple steps, each of which is performed by a specific hardware component; these components are arrange in a *pipeline* manner (the output of component i corresponds to the input of component $i+1$). Thus, when a component has delivered its output to the following one, it can start working on the next micro-instruction: though the global execution time for a micro-instruction fulfillment does not change, the instruction throughput grows significantly.

Pipelines are very important in Micro Architecture design (the *Mic-3* and *Mic-4* models proposed in [14] expose a three steps and a seven steps pipeline respectively). Unfortunately, the key concept of pipelines is the presence of *different* hardware components, that perform *different* tasks, making this approach not suitable for GPUs. Indeed, Micro Architecture level emulation is taken into account due to the chance of express task parallelism in terms of data parallelism, using the same program to perform different instructions. Thus, the introduction of different components (that must be concurrently executed) represents a backward step, that does not completely exploit the GPU parallelism.

This is a great limitation: it prevents GPU-based emulators to exploit

one of the greatest innovation of the corresponding targets processors. As a consequence, branch predictors too become useless. In a pipelined processor it is necessary to start fetching the new instruction when the current one has not finished yet, and if the current instruction is an either conditional or unconditional jump, it could be difficult to determinate the next instruction: a *Branch Predictor* is a further CPU component that tries to evaluate the branch that could be undertaken from a jump instruction, avoiding a CPU stall (i.e. the CPU waits until the current instruction is completed). Since GPU-based emulators cannot support pipeline, branch predictors become useless too.

Cache memory The use of cache memory is a great improvement in modern CPUs design: memory latency is too big if compared with registers access time, that cause the memory delays to be a bottleneck for processors' performance (several processor cycles are wasted while waiting for memory operations to be completed). The cache memory is a small and fast memory that contains frequently requested memory words. Thus, when a processor wants to read a word from the Main Memory, it firstly checks if the required word is stored in the Cache Memory and, in case of cache-miss, it forwards its request to the Main Memory. Due to the very small latency of cache memory, the less the number of cache-miss is, the higher performance the processor will reach.

Now the question is "*how the improvements gave by the presence of the Cache Memory can be exploited in the development of a GPU-based emulator?*" First of all, it is important to point out that GPU programming usually does not allow the rough hardware management, thus every memory operations must be performed using the Main Memory locations; secondly, one of the main GPUs feature is the high memory performance, in terms of low latency and high bandwidth; thirdly, GPUs memory exposes a hierarchical organization with multiple memory area with different sharing features and performance. Clearly, the more the number of cores sharing a memory area is, the lower the ap-

plication performances will be (due to the high number of cores that compete for the bus usage).

Thus, even if Cache Memory technique is not direct implementable in a GPU-based emulators, it is possible to achieve better performance exploiting different memory areas, using techniques similar to those relative to Cache Memory.

Reduction of execution path An ISA instruction is implemented by a number of micro-instructions that can vary from one to more that a dozen and for each of which the whole execution path has to be executed (with no pipeline). Furthermore, *Mic-1* forces every micro-instruction sequence to end with *main1*, whose only job is the increase of the PC register and the fetch of a new opcode. *Mic-2* model introduces a new hardware component called *IFU* (for *Instruction Fetch Unit*) that automatically increments the program counter at the end of each micro-instruction sequence, reducing the number of required micro-instruction for an ISA instruction execution. This approach is not direct implementable in GPU-based emulator, indeed, *IFU* is a separate component (i.e., it performs a different task), working in parallel with the rest of the processor and GPU cores cannot execute different tasks.

However it could be possible to increase the PC register during the execution path. This way, the emulation time required for a single micro-instruction become a little longer, but all ISA instructions are implemented with one less micro-instruction, reducing global execution time.

Another important point is the propagation delays for memory operations: as described in section 4.1.2, a memory operation issued in cycle i is performed during cycle $i+1$, making data read or written from cycle $i+2$; hence, a big number of cycle are wasted waiting for the memory operations fulfillment. In the best case, these cycle are used for other

purposes, but sometimes the complete execution path remains unused for a cycle.

Since a GPU based emulator mimics both the Main Memory and the registers with a fast Main Memory residing on the Graphic device, these hardware delays can be discarded, in order to make an operation completed at the end of the cycle that launched it. Moreover, *Mic-** models allow *fetch*, *read* and *write* operations, that always use MAR, MDR, MBR and PC registers as source and destination. It should be possible to implements memory operations that performs data exchange between the memory and arbitrary registers or between arbitrary memory locations; these operations are possible in current models, but require several cycles.

It is highly important to note that this section has described what can and cannot be implemented in GPUs with a pure *SIMD* design, chapter 3 describes the GPU programming and relative tools that usually implements a form of the *Single Program Multiple Data* paradigm (*SPMD*), that permits GPU cores to execute a limited number of different flows. This thesis investigates exactly this topic, the *ISA* level emulation (explained in chapter 5) forces each GPU core to perform a different process, while using the *Micro Architecture* level approach GPU cores run the same code. Hence, these two approaches could be coupled together, especially with regards to *Pipelines*, that are probably the best improvements in the Micro Architecture field. Although a many steps pipeline cannot be directed implemented in GPUs, it could be possible to design a limited-step pipeline, as long as GPU cores can execute different code.

Concluding, Micro Architecture designs have been widely improved during years with a lot of features, however not all of these features can be correctly replicated in a GPU-based emulator, forcing you to develop a legacy processor emulator with low performances. Beside this, GPUs offer some new interesting chances, that are not available in classic CPU hardware. The most important is the massively parallel computation power, that allow

to express the execution path in terms of multiple flows that operate on the same registers. Moreover, GPUs usually implement floating point operation in hardware, giving the chance to improve some ISA instructions implementation that have otherwise required multiple simpler micro-instructions.

6.3 The *MIC* model

The first Micro Architecture level emulator is a precise implementation of the *Mic-1* model described in [14].

As explained in chapter 3, a GPGPU application consists in two parts: a *host* program (a classical C/C++/Java application that runs on the CPU) and a *kernel*, the routine executed on the Graphic hardware. The *host* job is to compile the kernel code, to provide kernel arguments and to start the kernel execution. At run-time, no kernel-host communication is allowed so everything the kernel needs has to be provided before it starts.

In particular, there are three class of arguments:

Execution Environment It consists in Registers, Main Memory and Control Memory. In a parallel context with several parallel processes, there will be a Registers array for each process, a unique Main Memory array (with several *LV+Stack+Heap* areas, as described in section 4.1.2) and a unique read-only Control Memory array

Field selection Micro-instructions are composed of several bits grouped in fields, these fields have to be isolated for specific usages. Since fields are part of a 36 bits word (the micro-instruction), they can be isolated using a combination of *left-shift* and *mask* operations. Let mi be the micro-instruction, within kernel arguments there are two arrays, called `offset` and `mask`, such that the i-th field is obtained with:

$$field[i] = (mi >> offset[i]) \wedge mask[i] \tag{6.1}$$

Furthermore, some of these fields are used as they are coded in the micro-instruction (e.g. the ALU operation), while others act as a boolean flag (a single bit that can be either 0 or 1, like the *read* operation bit). Fields belonging to this second category require to be extended to a 32 bit word, as explained in section 6.3.1. Thus, there will be a further array within the kernel arguments, called `extension`. Its elements can be either 1 or 0xFFFFFFFF and each of them multiplies the corresponding field value, such that some fields remain unchanged, while single bit fields are extended to a 32 bits word. The resulting formulation will be:

$$field[i] = ((mi >> offset[i]) \wedge mask[i]) \times extension[i] \tag{6.2}$$

Registers selection An ALU operation is defined by the operation itself and by the registers involved in it, a micro-instruction contains indices that express references to registers. However, not all registers can be involved in an ALU operation, some registers are read-only and some other write-only. Micro-instruction fields express an index that may not match the real register index. Within the kernel arguments there are two arrays, *readable* and *writable* that act as a second level indexing. For example, $readable[i]$ holds the index of the i-th readable register, that could be different from the i-th register. Let C be the micro-instruction field for the *C Bus*, and R the result of an ALU operation, the code will be:

$$regs[writable[C]] = R \tag{6.3}$$

6.3.1 A branch-free code

One of the constraints of this work is the need for each GPU core to execute the same code. Indeed, Micro Architecture level is investigated exactly

for this purpose, it allows the emulator to execute the same program avoiding the use of a *switch*-like statement that call a different routine depending on the fetched opcode. However, though all GPU cores perform the same program, they can still attempt to execute conditional statements that could cause different code to be executed.

Hence, the emulator code must be completely *branch-free*, in order to avoid divergent executions. For this purpose, the emulator code mimics the hardware behavior.

A hardware circuit does not perform any branch, the basic component is the *bit* and bits are composed using logical operations like conjunction, disjunction and negation.

So, how to express different behaviors from bits and logical operators? At the hardware level, "behaviors" are just signals propagations, thus, a "branch" is a bits word that can be composed with a conditional words, that acts as a boolean value. Therefore, the computation must be expressed in terms of values.

Using this approach, the conditional statement

```
if(C) B1 else B2
```

has the form

$$X = (C \wedge B_1) \vee (\neg C \wedge B_2) \tag{6.4}$$

Since the branches B_1 and B_2 are just values, they can be composed using operators $\wedge$, $\vee$ and $\neg$, obtaining the value X, that corresponds to an intermediate step of the computation.

Clearly, the expression of the whole emulator code in terms of values would require a big coding effort, thus this technique is used only to avoid branches.

For example, a *read* operation copies a value from the memory location pointed to by the *mar* register to the *mdr* register. When this operation is requested, the second bit of the *Mem* field is turned on and the code

```
if(readBitIsOn)
    mdr = mem[mar]
```

is translated in its branch-free version

$$mdr = (readBitIsOn \wedge mem[mar]) \vee (\neg readBitIsOn \wedge mdr) \tag{6.5}$$

Clearly, using 32 bits words, conditional guards (the *readBitIsOn* in the example) must be 32 bits long too, otherwise the bit-wise composition would fails. For this reason, fields extracted from the micro-instruction that consists of a single bit (e.g. memory operation bits), have to be extended to 32 bits words. This is achieved using the `extension` array, whose elements multiplies micro-instruction fields either by 1 or 0xFFFFFFFF, depending on the field usage.

Although this technique allows the complete removal of all branches from a program, there is an important drawback: a value is always written in a memory location, while this is not true using an *if*-statement.

This is particular important when considering several processes running in parallel, sharing a memory area. Parallel processes are designed in order to avoid unconsistent memory views, so if for example a process writes a value in a memory location, another process will neither write to nor read from the same location. However, in a branch-free code "do not write" means "write the same value that was previously stored". Thus, it could be possible that a process A writes a value in a memory location and another process B do some other computation. When B execution reaches the "write" code, it performs a useless write operation. The problem arises when the registers of both processes A and B point to the same memory location, regardless the fact that from the B point of view, its write operation has no effects on the environment.

Thus, it is highly important to ensure that a idle process registers do not refer to any shared memory location, in order to avoid processes interferences.

Considerations

The complete removal of branches allows to execute the code on a pure *SIMD* multiprocessor avoiding the divergent execution. However, it produces a performance downgrade.

Since any conditional statement implies the computation of two values, one of which will be always discarded. Consider for example the branch-free version of the read operation, expressed in equation 6.5, at any cycle two conjunctions, one negation an one disjunction are performed, regardless the fact that read operation could or could not be required.

As a more significant example, consider the ALU emulation. Starting from two input A and B, it perform four distinct results:

- $A \wedge B$
- $A \vee B$
- $A + B$
- $\neg B$

but only one of these will be selected and stored in a register. Although the wasted computation is not too big, this fact poses a limitation for future ALU capabilities. An improved version of this work could include a modern ALU, able to perform difficult operation in a single cycle (e.g. integer multiplication, division and reminder).

6.3.2 An exactly mapping

The *MIC* emulator mimics the behavior of the *Mic-1* model described in [14], it consists in a loop that iterates the following steps:

1. it selects the micro-instruction pointed to by the *MPC* register within the Control Memory
2. it selects the input register that writes on the B bus (the A bus always receives data from the H register)

3. it selects the output registers that receive the ALU results
4. it performs the ALU operation (keeping note of an either *null* or *negative* result)
5. it stores the result
6. it performs a fetch operation
7. it performs a read operation
8. it performs a write operation
9. it computes the next micro-instruction to be performed, according to the *PC* register and the null/negative flags computed in step 4

Memory operations issue

Steps 6, 7 and 8 refer to the memory operations, the proposed design is very inefficient, since it mimics the exactly hardware behavior, causing a performance downgrade, for example:

- These steps are executed one after the other, while a real hardware implementation would perform them in parallel
- Since *read* and *write* are mutually exclusive, at most one of them can be requested
- The delayed execution of *Mic-1* is kept, if a memory operation is required in cycle *i*, it is performed during cycle *i+1* and completed from the beginning of cycle *i+2* (see section 6.1.3)

An efficient implementation of the *Mic-1* model should abstract from hardware details making a memory operation completed at the end of the cycle that has requested it. Moreover, since each step consists in a value copy between two memory locations, it could be possible to use the same code varying source and destination value

However, the *MIC* emulator is a precise mapping of the *Mic-1* model, so this details have been kept. The improved version *PMIC* address these issues.

6.4 The *PMIC* model

The improved models described in section 6.2 are taken into account, leading to the development of the *PMIC* emulator.

6.4.1 A parallel architecture

The first feature that has been considered for Micro Architecture level emulation is the *Pipeline*, a CPU cycle is divided into several steps that are performed by specific hardware components, arranged in a pipelined manner (i.e., the output of the component i matches the input of the component $i+1$). Since components are autonomous, after one component has executed its task it can immediately start working on the next cycle, without having to wait the whole cycle conclusion. Hence, although the required time for one cycle execution remains unchanged, the instructions throughput increases.

The pipeline emulation is not a trivial task, since it requires that several parallel processing elements mimic the various hardware components behavior and the *SIMD*-like architecture of Graphic devices forces its processing elements to execute the same instructions stream simultaneously.

Nevertheless, although it is not possible to split the emulation of a micro-instruction into several steps, the parallel power of graphic hardware can be exploited emulating the execution of multiple instructions in parallel. Thus, the emulation environment will be composed of several *Compute Units* (*CUs*), which work on the same registers set. For sake of consistency, several *CUs* can process data coming from the same register, but the output data must be delivered to different registers.

Micro-instructions are composed in order to implement an ISA instruction (see section 6.1.5), now this implementation can be expressed with multiple

parallel flows. It is highly important to note that while an N steps pipeline could improve the instructions throughput by N times, an emulator with N parallel flows does not reach the same improvement. Micro-instructions are often related by a causal relationship, thus some of them must be executed in different steps. For example, the code

```
pc=pc+1; fetch;
h=mbr << 8; ;
pc=pc+1; fetch;
h=h | mbr; ;
pc=cpp+h; fetch;
```

performs a very common task, it reads a two-bytes operand from the *Text* area and uses it as an offset inside the *Constant Pool Portion*, whose base position is pointed to by the *CPP* register (a *fetch* operation copies the byte pointed to by the *PC* register into the *MBR* register, since only one byte at a time is fetched, a two bytes operand requires the first one to be fetched and left-shifted and then it is possible to fetch the second one).

Since each micro-instruction depends on the result produced by the previous one, they cannot be executed in parallel.

Despite this example, the presence of several parallel *Compute Units* usually produces a good performance improvement when coupled with some additional features:

General-Purpose and Constant registers Special-Purpose registers are designed in order to support a single execution flow. Sometimes two independent (i.e., not related by a casual relationship) micro-instructions must be serializes because there are not enough available registers to support both computations. Thus, the presence of some General-Purpose register permits to better exploit the parallel architecture. Furthermore, some constant registers are introduced, they hold the constant values 2, 3, 4, 5 in order to avoid the computation of these values every time they are required (constant values 0 and 1 are im-

plicitly provided by the ALU).

No *main* micro-instruction The implementation proposed in [14] concludes each micro-instructions sequence with a jump to the *main* micro-instruction, whose only job is the *Program Counter* increment, so that a new opcode can be fetched and a new instruction can start. Now, this job can be executed in parallel with the rest of the sequence (using a dedicated *Compute Units*). Hence, each sequence of the model proposed in [14] is now reduced by one.

Prefetching Some ISA instructions require one or more operands to fulfill the execution and the need for an explicit *fetch* for each of them increases the average number of required micro-instruction. To overcome this problem, operands must be available in some registers without having to be explicitly fetched. The *fetch* operation has been changed in order to copy a whole 32 bits word instead of a single byte. The opcode and the possible operands can be accessed through the virtual registers *MBR0*, *MBR1*, *MBR2*, *MBR3* respectively, that correspond to the various bytes within the *MBR* register. If an instruction requires more than three operands, they have to be explicitly fetched.

Memory operations immediately performed The need for two cycles for a memory operation fulfillment is due to the signals propagation time and the Main Memory latency. Since an emulator can avoid to deal with these hardware details, memory operations can be concluded in the same cycle when they have started, reducing the global number of cycles.

The *PMIC* emulator uses three parallel *Compute Units* and it theoretically reduces the average number of cycles by 2.56 times w.r.t. the sequential implementation proposed in [14]. In the practice this value become 1.9.

A *Computation Unit* can be considered *exploited* if its usage reduces the number of cycles. Hence, in the example above, five micro-instructions are

related by a causal relationship and no additional *CU* can be exploited, since the number of cycle still remain five.

The number of *three* has been chosen as the number of *Computation Units* for the *PMIC* model because a lot of micro-instruction sequences exploit the presence of three *CU*, however a further one would not produce better performances.

Moreover, the presence of three *Compute Units* allows to map each memory operation to a dedicated *CU*. In this way all *fetch* operations will be performed by the first *CU*, all *read* operations by the second one and all *write* operations by the third one, regardless of the order by which operations are requested. Therefore, the emulation time per cycle can be reduced since at each cycle only one operation will be performed instead of three.

6.4.2 A new micro-instruction structure

A micro-instruction is a set of bits that defines a particular hardware configuration. Thus, the more complex the hardware is, the more expressive the micro-instruction must be and the one described in section 6.1.1 is no longer expressive enough for the *PMIC* computation model.

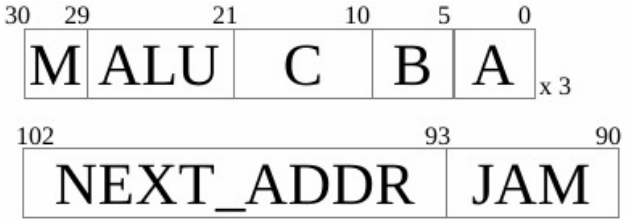
A micro-instruction is composed of three parts: the *ALU Computation*, the *Memory Operation* and the *Next Address Computation*. Since the *PMIC* model exposes three *Computation Units*, the first and the second micro-instruction parts must be replicated three times. The third part consists in the computation of the address of the next micro-instruction to perform and all *Computation Units* must agree on that address. Therefore, the third part of the micro-instruction is the same for all *CUs*.

The *ALU Computation* part has to be refined since the *PMIC* model has some additional register, the new versions of the *B Bus* and the *C Bus* fields must be 5 and 11 bits long respectively, since there are 25 readable registers and 11 writable registers. Furthermore, the *PMIC* model makes the ALU left input bus to be programmable while the previous micro-instruction structure does not allows to control it. Thus, an exact copy of the *B Bus* field must

be added to the structure for the *A Bus*.

In the old version of the micro-instruction structure the *Memory Operation* part is a three bits block such that each bit denotes a memory operation. Since each *CU* performs a specific memory operation, the corresponding part of the new structure should be a single bit. Both the *ALU* part described above and this *Memory* bit are replicated for each *CU*.

Figure 6.3: The *PMIC* micro-instruction structure



Finally, the *Next Address Computation* part remains unchanged. The *PMIC* micro-instruction structure is depicted in Figure 6.3, it is a 102 bits word while the simple *MIC* model described in [14] requires only 36 bits.

The *PMIC* emulator implements it using four distinct 32 bits words, the first three have the same structure, each of them denotes the *ALU* and the *Memory* parts of a single *Compute Unit*. The last word corresponds to the *Next Address Computation* part. Since this last part is composed of 12 bits, a 16 bits variable should be enough for holing it, however almost all GPU programming libraries do not allow to write in variable smaller than 32 bits. Thus, the GPU implementation of the micro-instruction requires 128 bits (four 32 bits words).

6.4.3 A language for PMIC microcode

Since micro-instructions are just blocks of bits, it could be possible to write them using constant values, shifted by a correct offset. For example, the micro-instruction

```
mar=sp+1 ; read ; goto main
```

can be written as

```
micro_instr = REGISTER_SP << BUS_A_OFFSET
            | REGISTER_NULL << BUS_B_OFFSET
            | A_PLUS_ONE << ALU_OFFSET
            | REGISTER_MAR << BUS_C_OFFSET
            | READ_OPERATION << MEM_OFFSET
            | main_index << NEXT_ADDR_OFFSET;
```

This technique has been adopted for the *MIC* version of the emulator and it has proved to be very tedious and error-prone. For this reason a high level language for the *PMIC* model and a corresponding compiler have been developed in order to facilitate micro-instructions writing and maintenance.

This language allows to define ISA instructions implementation given the mnemonic opcode name (*add*, *swap*, *goto*, ...). Each ISA instruction contains one or more micro-instructions, each of which is composed of three parts as described in the previous section, *ALU*, *Memory Operation* and *Next Address*; first two parts support up to three parallel flows while the third one is common for all *Compute Units*. The following grammar defines the language.

⟨*isa-instr*⟩ ::= ⟨*opcode-name*⟩ '`{`' ⟨*microcode*⟩ '`}`'

⟨*microcode*⟩ ::= ⟨*micro_instr*⟩
| ⟨*microcode*⟩ '`;`' ⟨*micro-instr*⟩

⟨*micro-instr*⟩ ::= ⟨*alu*⟩
| ⟨*alu*⟩ '`:`' ⟨*mem*⟩
| ⟨*alu*⟩ '`:`' ⟨*addr*⟩
| ⟨*alu*⟩ '`:`' ⟨*mem*⟩ '`:`' ⟨*addr*⟩

⟨*alu*⟩ ::= ⟨*one-alu*⟩
| ⟨*alu*⟩ '`//`' ⟨*one-alu*⟩

⟨*mem*⟩ ::= ⟨*one-mem*⟩
| ⟨*mem*⟩ '`//`' ⟨*one-mem*⟩

Since registers access mode can be either read-only, write-only or read-write, there are two distinct syntactic categories, the ⟨*regR*⟩ one and the ⟨*regW*⟩ one, for readable and writable registers respectively. Moreover, readable registers can be composed using a defined set of *ALU* operations, defining the ⟨*R* − *value*⟩ category and the result of an *ALU* computation can be delivered to multiple comma-separated writable registers, that define the ⟨*L* − *value*⟩ category.

ALU operations are provided via mnemonic names, each of which has a corresponding constant value. The language grammar will be:

⟨*one-alu*⟩ ::= ⟨*L-value*⟩ '`=`' ⟨*R-value*⟩

⟨*L-value*⟩ ::= ⟨*regW*⟩
| ⟨*L-value*⟩ '`,`' ⟨*regW*⟩

⟨*R-value*⟩ ::= '`zero`' | '`one`' | '`minus-one`'
| ⟨*regR*⟩ '`<< 8`'
| ⟨*regR*⟩ '`>> 8`'
| '`not`' ⟨*regR*⟩
| ⟨*regR*⟩ '`and`' ⟨*regR*⟩
| ⟨*regR*⟩ '`or`' ⟨*regR*⟩
| ⟨*regR*⟩ '`+ one`'
| ⟨*regR*⟩ '`+`' ⟨*regR*⟩
| ⟨*regR*⟩ '`+`' ⟨*regR*⟩ '`+ one`'
| '`-`' ⟨*regR*⟩
| ⟨*regR*⟩ '`minus one`'

Memory operations are declared through the corresponding mnemonic names *fetch*, *read* and *write*. This part is optional, if it is not specified, no operation will be performed.

⟨*one-mem*⟩ ::= '`fetch`'
| '`read`'
| '`write`'

Finally, the *Next Address Computation* part defines which micro-instruction will be performed after the current one. In most cases, within a micro-instructions sequence, each one "invokes" the next one and the last one "invokes" the micro-instruction whose index is stored in the *PC* register. This is the common behavior, that is performed when *Next Address Computation* part is omitted. Different behaviors raise in presence of conditional and unconditional jumps, the first one depends on the either *null* or *negative* result of an *ALU* computation, while the second one fetch an new operands from an arbitrary part of the memory.

It is important to note that a conditional jump is defined by two micro-instructions (one for each branch) whose opcodes must differ only for the 9^{th} bit. When a micro-instruction has the either *null* or *negative* jump bit on, the 9^{th} bit of the *NEXT_ADDR* field is reversed according to a null or a negative *ALU* result respectively. Thus, the definition of a conditional jump does not allow to specify two arbitrary micro-instructions for the two branches (like an *if-then-else* statement). Within the language grammar, the syntactic category $\langle flag \rangle$ define the condition (n for *Negative* and z for *Zero*) and the $\langle opcode - name \rangle$ specifies the micro-instruction to be performed if the condition is true. If the condition is false, the $\langle opcode - name \rangle$ index is *XOR-ed* with the value `0x100` in order to reverse the 9^{th} bit.

⟨*addr*⟩ ::= '`next`' ⟨*opcode-name*⟩
| '`next (`' ⟨*flag*⟩ '`)`' ⟨*opcode-name*⟩

⟨*flag*⟩ ::= '`n`' | '`z`'

A language compiler has been developed using the *Flex* lexical analyzer and the *Bison* parser generator tools [19], which are both part of the *GNU* project. Micro-instructions are encoded with the format used by the emulator

and written to the disk as a file. Thus, the emulator does not have to rebuild all micro-instructions at every run (as the previous emulator version do), but it just read the file the compiler has built, coping the *Control Memory* (the set of all micro-instruction) to the GPU buffer.

An example

This example is extracted from the *PMIC* implementation used in this thesis, the ISA instructions *add* and *if_cmpeq* are implemented using two and three micro-instructions respectively (the original sequential implementation proposed in [14] requires four and seven respectively).

The *if_cmpeq* sequence ends with a conditional jump, for this purpose the two micro-instructions *true* and *false* are introduced. The former fetches a 2 bytes operand ad uses it as an offset for the *PC* register, while the latter just skips to the following opcode. Clearly the opcodes for *true* and *false* are equal except from the 9^{th} bit, thus the conditional jump "$next(Z)true$" means that a *null* result forces to jump to the *true* micro-instruction, while a *non null* result makes the control to move to ($true \oplus$ `0x100`), that is $false$ (the $\oplus$ operator is the bitwise *XOR*).

```
add {
       mar,sp = sp-one
       : read;

       mdr,tos = mdr+tos // pc = pc+opl
       : write // fetch
}

if_cmpeq {
       mar = sp-const2 // h = tos
       : read;

       tos = mdr // mar = sp-one
```

```
        : read;

        z = h-mdr // sp = sp-const2
        : next(Z) true
}

true {
        h = mbr1<<8 | mbr2;

        pc = pc+h
        : fetch
}

false {
        pc = pc+opl
        : fetch
}
```

Chapter 7

Conclusions and Future Works

Previous chapters described two different approaches for the emulation of a parallel environment using a GPU as the *host* platform. Since the *ISA* level emulation technique (the classic one) presents a non trivial divergence issue, a new emulation approach has been introduced in order to overcome the limitation of the classic mode. Besides this, the *General-Purpose Computation on Graphics Devices* (*GPGPU*) has been studied in order to understand how GPU programming can be achieved, which programming libraries are currently available and how these libraries behave. Indeed, GPGPU libraries usually spread the execution units among GPU cores according to some internal policies in order to limit the divergence phenomena as much as possible.

Our experiments evaluate the emulator performances with N different parallel processes; the classic emulation approach would theoretically result in a linear growth of the execution time with higher values of N (due to the divergence phenomena). However, the *OpenCL* library arranges the execution so that the time growth is very small. For this reason the classic emulation approach is not completely discarded and it has been compared with our new technique that produces better timing performances at the cost of a more complicated emulator design.

This new emulation approach has been implemented in two distinct ver-

sions, called *MIC* and *PMIC* respectively; while the first one emulates one instruction at time the latter provides three distinct *Processing Elements* for each execution flow. Clearly the *PMIC* model is faster (in a single cycle, it can perform more work) but it is able to support a limited number of parallel process.

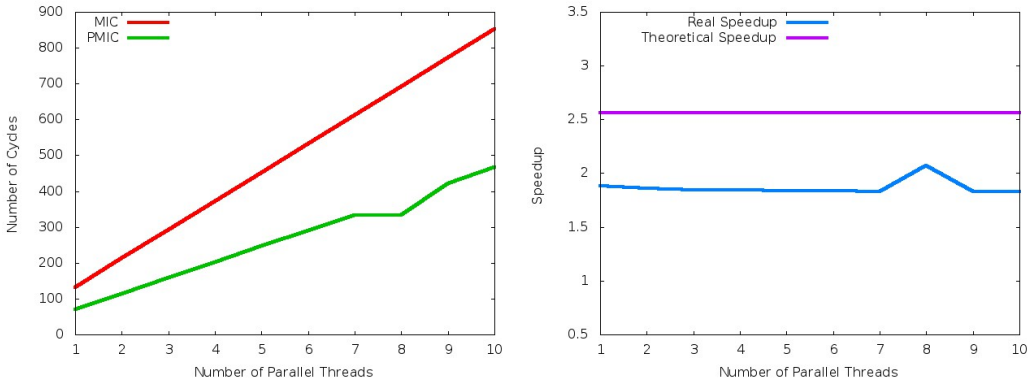
This is due to the fact that the *OpenCL* library establishes the maximum number of parallel execution flows according to both the device capability and the complexity of the *kernel* routine. Let N be the number of available *Processing Elements* exposed by the Graphics Device, the *MIC* emulator has a rather simple design and so it supports up to N parallel threads. On the contrary, the *PMIC* emulator is more complicated and it can exploit few processing elements. Moreover, since the *PMIC* model exposes three distinct *Processing Elements* for each emulated process, the reduced power has to be partitioned among processes.

For example, the *Radeon RV710* GPU (the testbed device used in this thesis) exposes 128 *Processing Elements* but the *PMIC* kernel can be launched only on 32 parallel threads (due to its complexity); the number of 32 has to be in turn partitioned among emulated processes. Thus the *PMIC* emulator cannot launch more then 10 parallel execution flows, while the *MIC* emulator can scale up to 128 threads.

For sake of clarity, the *OpenCL* library would be able to support an arbitrary number of parallel execution units (*Work Items* in the OpenCL nomenclature); to achieve this, *Work Items* are clustered in several *Work Groups* that are interleaved on the device *Processing Elements*. The problem is that *Work Items* from different groups do not have a consistent memory view (see section 3.3.2). Thus, this thesis limits the emulation scalability to the number of available processing elements of the adopted device (i.e., it will be created only one group).

The introduction of three parallel *Processing Elements* for each process emulation makes the *PMIC* model to reduce the required number of micro-instruction to be performed for an ISA instruction fulfillment. Consequently,

Figure 7.1: Execution Time



(a) (b)

the global execution time decreases too. The theoretical speedup obtained by the *PMIC* emulator is 2.56 but in the practice this value become 1.88; this is due to the fact that the ISA instructions with a high improvement factor are more rare, while the most frequently used instructions expose a limited improvement factor. Figure 7.1(a) depicts the number of emulated clock cycles with an increasing number of emulated processes. In the rightmost plot of the Figure the blue line represents the ratio of cycles amount obtained by both emulators at a given number of processes; these values average is 1.88 and it is compare with the theoretical speedup of 2.56 (the purple line).

These three emulation approaches ("*ISA*", "*MIC*" and "*PMIC*") are evaluated according to two measurements:

- Execution time
- Graphics Device memory usage

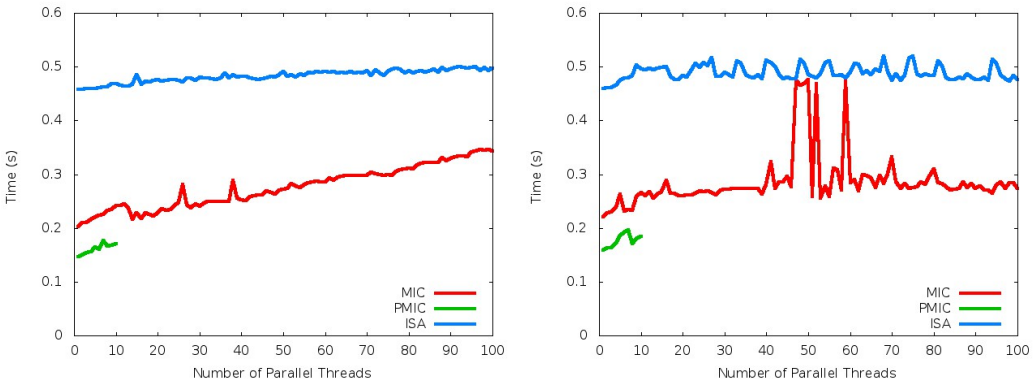
Following sections expose some results of both evaluations, after which the thesis will conclude with a suggestion for a hybrid solution, in order to improve the performance of both approaches. The evaluations come from two distinct executions, that emulate the same process and several distinct processes respectively. It is important to claim that the former experiment

does not make GPU cores to proceed the same instruction sequence synchronously; since the same process is launched at different temporal steps, the GPU *Processing Elements* execute the same instruction sequence but not the same instruction at the same time.

7.1 GPU execution time

Section 4.2.1 describes this type of evaluation: different processes could be parallel emulated and the resulting time would be a function of both the number and the complexity of the processes. This thesis wants to investigate only the emulation scalability, thus in addition to the *Execution Time*, the *Time per Instruction* has been measured.

Figure 7.2: Execution Time

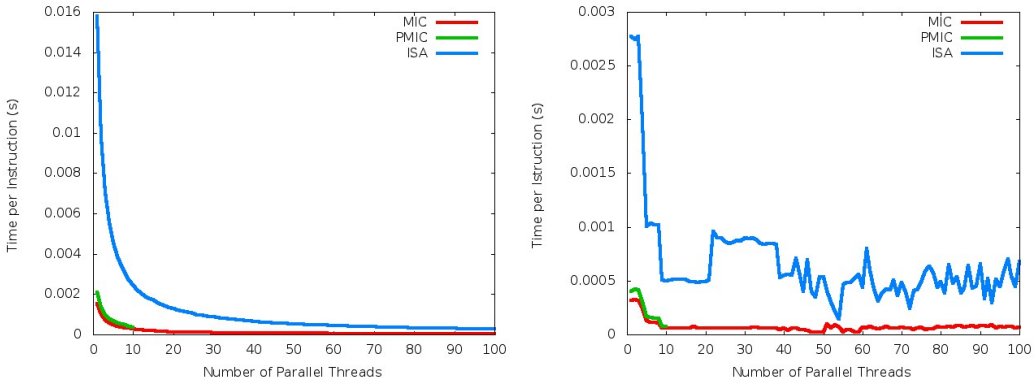


(a) (b)

Figures 7.2 and 7.3 show the *Execution Time* and the *Time per Instruction* plots respectively; in both pictures the leftmost graph refers to the emulation with the multiple instances of the same process, while the rightmost one shows the results of different processes concurrently emulated.

All graphs show three lines, the blue one is the trend of the classic emulation approach, called "*ISA*" while the red and the green represent the two versions of our new emulation technique called "*MIC*" and "*PMIC*" respectively. As said before the *PMIC* model cannot scale up to 10 execution flows,

Figure 7.3: Time per Instruction



(a) (b)

that is the reason because the green line stop soon.

These graphs show that both approaches scale well as the number of emulated processes grows; Figure 7.2 (a) depicts that the *MIC* approach requires less time to be executed but has a bigger growth rate then the *ISA* approach; this observation suggests that with a bigger number of parallel processes both techniques will require the same time.

The rightmost plots (in both Figures 7.2 and 7.3) show a similar trends except from the irregular trend due to the difference between the emulated processes.

7.2 Memory usage

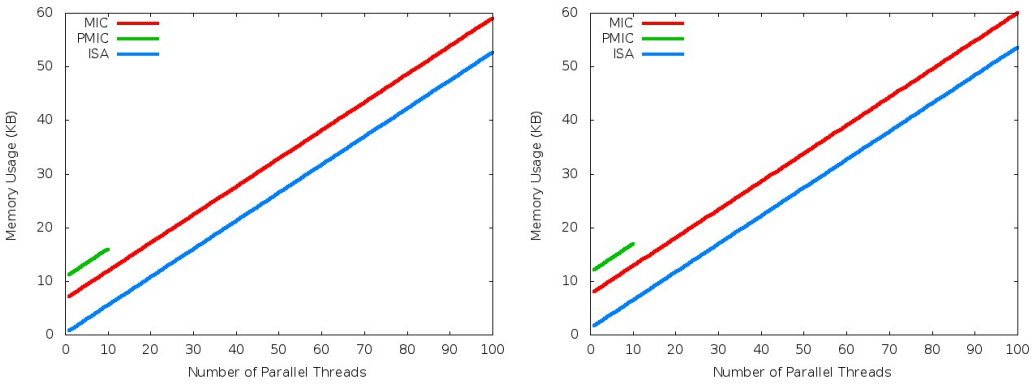
GPGPU programming largely depends on the Memory utilization, a correct usage of the GPU memory hierarchy can increase the application performances; for this thesis purposes different memory approaches have been tested, in particular the emulators have been developed using both the *Local* memory, the *Global* memory and a combination of them. The best performance has been achieved using the *Global* memory; although it is the slowest memory area, it is the easiest way for processes synchronization.

All the adopted models assume a theoretical memory capacity of 4 GB,

this is not true in practice since this value must deal with the Graphics device memory capacity; the *Radeon RV710* GPU [3] (the testbed for this thesis experiments) exposes 1 GB of memory, that stores the emulator memory and some additional data structures required by the emulation.

Due to the limited memory capacity of a low-end GPU and the big impact that the memory usage has in a GPGPU application, the required data structures amount for each emulation approach has been measured in order to evaluate the memory requirement of an emulation technique.

Figure 7.4: Memory Usage



(a) (b)

Both experiments show the same trend: the classic emulation approach (the blue line) is the cheapest solution for memory usage; indeed the *Micro Architecture* level emulation is possible because the code is translated into data.

7.3 Future Works

The plot of Figure 7.2(a) highlights three facts:

- The execution time of the *ISA* emulator does not linear grows as the number of emulated process grows

- The *MIC* emulator produces better time performances then whose of the *ISA* emulator
- The *MIC* emulator exposes a growth rate bigger then the one of the *ISA* emulator

Thus we can conclude that the *Micro Architecture* level emulation introduces some important features, but its capabilities are very limited since the emulated processor is rather legacy; on the other hand an *ISA* level emulator can exploit some of the modern processors features.

The *ISA* emulator performances highlight that different execution flows can be correctly managed by the *OpenCL* library, thus it could be possible to implement in a *MIC*-like emulator some of the *Micro Architecture* improvements described in section 6.2. The *pipeline* is one of the great improvement in the processor design field and the use of a pure *SIMD* architecture does not allow to implement a pipeline. The *ISA* emulator results contradict this strict requirement, thus the pipeline components can be emulated via several GPU *Processing Elements*.

This pipelined version of the *MIC* emulator must take into account that the complexity of the kernel has a big impact in the maximum number of parallel execution units the GPU can run, as the *PMIC* model highlights the scalability of an approach can be highly reduced by the kernel complexity, so a pipelined emulator cannot support a huge number of steps.

Bibliography

[1] Duncan R, *A survey of parallel computer architectures* . Computer (vol 23), February 1990.

[2] Javier Diaz, Camelia Munoz-Caro, Alfonso Nino *A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era* IEEE Transactions on Parallel and Distributed Systems (vol 23), August 2012

[3] GPUReview.com: Radeon RV710 GPU specification *http://gpureview.com/ati-rv710-chip-158.html*

[4] Michael Macedonia *The GPU enters Computing's Mainstream* Computer (vol 36), October 2003

[5] Dave Shreiner, The Khronos Opengl Arb Working Group *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1* Addison-Wesley Professional, 2009

[6] David Luebke, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papakipos, Ian Buck *GPGPU: general-purpose computation on graphics hardware* Proceedings of the 2006 ACM/IEEE conference on Supercomputing (Article No. 208) 2006

[7] Akira Nukada, Yutaka Maruyama, Satoshi Matsuoka *High performance 3-D FFT using multiple CUDA GPUs* Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units (Pages 57-63) 2012

[8] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, Parthasarathy Ranganathan *GViM: GPU-accelerated virtual machines Published by ACM 2009 Article* Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing (Pages 17-24) 2009

[9] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, Emmett Witchel *PTask: operating system abstractions to manage GPUs as compute devices* Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (Pages 233-248) 2011

[10] Tong Li, Brett P., Knauerhase R., Koufaty D., Reddy D., Hahn, S *Operating system support for overlapping-ISA heterogeneous multi-core architectures* 2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA) January 2010

[11] Shivani Raghav, Andrea Marongiu, Christian Pinto, David Atienza, Martino Ruggiero, Luca Benini *Full system simulation of many-core heterogeneous SoCs using GPU and QEMU semihosting* Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units (Pages 101-109)

[12] Hwu, Wen-mei, Kurt Keutzer, Timothy G. Mattson *The concurrency challenge* Design & Test of Computers, IEEE 25, no. 4 July,August 2008

[13] Dagum, L., Menon, R *OpenMP: an industry standard API for shared-memory programming* Computational Science & Engineering, IEEE (Volume:5 , Issue: 1) January,March 1998

[14] Andrew S. Tanenbaum, *Structured Computer Organization*, Prentice Hall, chapter 3 2012 (6th edition)

[15] Snir Marc, S. W. Otto, D. W. Walker, J. Dongarra, S. Huss-Lederman, *MPI: the complete reference* MIT press, 1995.

[16] R. Davoli, M. Goldweber *Virtual Square: Users, Programmers & Developers Guide* Lulu Books, 2011

[17] F. Bellard, *QEMU, a Fast and Portable Dynamic Translator* In USENIX Annual Technical Conference, FREENIX Track, pp. 41-46 2005.

[18] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for parallel programming* Pearson Education, 2004.

[19] J. Levine, *Flex & Bison*, O'Reilly, 2009

[20] J. Lee et al. *An OpenCL framework for heterogeneous multicores with local memory* In Proceedings of the 19th international conference on Parallel architectures and compilation techniques, pp. 193-204. ACM, 2010.

[21] CUDA Zone, *http://www.nvidia.come/object/cuda_home_new.html* October, 2011

[22] M.J. Flynn, *Some computer organizations and their effectiveness*, Computers, IEEE Transactions on 100, no. 9 1972

[23] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, T. J. Purcell, *A Survey of General-Purpose Computation on Graphics Hardware*, In Computer graphics forum, vol. 26, no. 1, pp. 80-113. Blackwell Publishing Ltd, 2007.

[24] B. Neelima, and P. S. Raghavendra, *Recent trends in software and hardware for GPGPU computing: a comprehensive survey*, In Industrial and Information Systems (ICIIS), 2010 International Conference on, pp. 319-324. IEEE, 2010

[25] http://www.gpgpu.org

[26] M. Pharr, R. Fernando, *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*, Addison-Wesley Professional, 2005

[27] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, *Brook for GPUs: stream computing on graphics hardware* ACM Transactions on Graphics (TOG), vol. 23, no. 3, pp. 777-786. ACM, 2004

[28] Khronos OpenCL Working Group, *The opencl specification*, A. Munshi, Ed, 2008

[29] M. Goldweber, R. Davoli, M. Morsiani, *The Kaya OS project and the $\hat{I}\frac{1}{4}$ MPS hardware emulator*, In ACM SIGCSE Bulletin, vol. 37, no. 3, pp. 49-53. ACM, 2005.