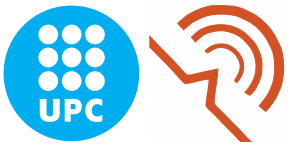

Universitat Politècnica de Catalunya
Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

Design and Implementation of a PTX Emulation Library

by

Albert Claret Exojo

Advisor: Isaac Gelado Fernández

Barcelona, July 2009

Dedicado a mis padres y a Susan, por su paciencia, cariño y apoyo.

Contents

Acknowledgements

Ante todo, quiero agradecer a Isaac Gelado su paciencia, ayuda y apoyo durante estos meses. A pesar de mis infinitas dudas y preguntas, no le he escuchado quejarse ni una sola vez.

También quiero agradecer el apoyo y la paciencia a mi madre, que probablemente se alegrará tanto o más que yo de que haya acabado la carrera, a mi padre y a Susan. Sin vosotros, no lo hubiera logrado.

Y por último, agradecer el apoyo de mis amigos, que me han aguantado y, por algún extraño motivo, siguen aguantando todos los días. Gràcies als *Budgets*, als de *R3*, als de *DAT*, als de les altres plantes de l'Edifici Omega, a la gente del *C.EE.T.*, a *Las Niñas*, a los de *La Salle*, a los pibes del *ITBA*, al resto de los amigos que dejé en la Argentina, a los compañeros del *C6-103* por ayudarme durante estos meses y a mis viejos amigos del *ASB*.

Albert Claret

Chapter 1

Introduction

1.1 Motivation and Objectives

Intel co-founder Gordon E. Moore observed in 1965 that transistor density, the number of transistors that could be placed in an integrated circuit per square inch, increased exponentially, doubling roughly every two years. This would be later known as *Moore's Law*, correctly predicting the trend that governed computing hardware manufacturing for the late 20th century.

For many decades, software developers have enjoyed a steady application performance increase due to continuous hardware improvements as described by Moore's Law, as well as computer architecture improvements. Currently, however, the *memory wall*, which refers to the increasing speed difference between the CPU and memory, and the instruction-level parallelism wall (*ILP wall*), which refers to the inability to find more operations in an application which can be performed simultaneously due to data dependency, have been reached. Application performance no longer benefits from continuous processor frequency increases as it had before. Furthermore, other issues such as wire delays and static and dynamic power density prevent significant processor frequency increases [1].

The High-performance Computing (HPC) community has developed several strategies to work around these limitations [2]:

- Multi-core systems, where a processor includes several independent cores in a single integrated circuit. This strategy currently serves as the most viable candidate to uphold Moore's Law. Despite this, applications that heavily rely on sequential calculations would only see major advances if the processor frequency increased several orders of magnitude.

- Specialized processors, which allow a performance boost in a specific area. Digital Signal Processors (DSPs) are a popular example of specialized processors. They are optimized for quickly performing mathematical operations on a continuous set of data, such as multiply-accumulate instruccions and parallel address register computations. They are used extensively in TVs, cell phones and other embedded devices. Another popular example are Graphics Processing Units (GPUs), originally specialized in 3D graphics calculations.
- Heterogeneous computing architectures, which combine traditional and specialized processors to work cooperatively.

HPC Applications usually contain code that could benefit from specialized processors but also code that should be run in conventional processors. Heterogeneous platforms allow for each type of code to run in the processor best suited for the task, achieving accelerations of up to 100 times what a scalar processor can achieve independently [2]. Several types of heterogeneous platforms exist, such as those based on FPGAs, IBM Cell processors or GPUs. These are also called *accelerator architectures* [3], as they bring a big advantage in performance, cost or power usage over a general-purpose proces

Field-Programmable Gate Arrays (FGPAs) have been around for more than a decade, but recently, there has been increasing interest in them to be used as reconfigurable coprocessors. They can provide huge speedups over conventional processors in many applications [4] by allowing the creation of a custom instruction set and allowing data parallelism by executing the instructions concurrently on thousands of data pieces. Recently, field programmable object arrays (FPOAs) have been proposed as an alternative to FPGAs, allowing operation at higher frequencies, currently up to 1GHz. FPGAs have several major drawbacks that do not make them suitable for some applications. They usually do not have native support for floating point operations, rely on the designer to incorporate components such as memory or I/O, and require a design cycle much longer than traditional software applications [4].

Several heterogeneous computing systems can be linked to create a cluster, obtaining significant advantages in HPC applications. A hybrid system cluster can perform significantly faster than a similar-sized general purpose processor cluster [4]. A clear example of this is the *IBM Roadrunner*. As of July 2009, it is the world's fastest supercomputer, achieving 1.105 petaflop/s [5]. It is a hybrid design with 12,960 IBMPowerXCell 8i and 6480 AMD

Opteron dual-core processors [6].

The IBM Cell combines an in-order PowerPC core that controls eight simple SIMD cores (Single Instruction Multiple Data, which refers to a technique where a large group of simple processors perform the same task simultaneously, each of them with different data), called synergistic processing elements (SPEs). Each of them contains a Synergistic Processing Unit (SPU), local memory, and a memory controller [7]. The first major commercial application was its usage in Sony's PlayStation 3 gaming system, which brought world-wide attention to the chip.

The Cell architecture offers huge potential computing performance, but it does not come free. Existing code can not simply be recompiled, because it would all run in the main PowerPC, providing no speedup. Each SPE has its own local memory where code and data coexist. SPE loads and stores can only access the local memory, depending on explicit Direct Memory Access (DMA) operations to move data from the main memory to the local SPE store [7]. On the other hand, there are strong constrains on memory access alignment, as they have to be aligned to 128 bit boundaries. Writing vectorizing code, which performs the same operation on a set of data concurrently, is particularly hard. These features make the Cell a difficult platform to develop for, requiring expert programmers to hand-craft applications to achieve considerable speedups.

An alternative to Cell-based heterogeneous systems are GPU-based systems. General-purpose computing on graphics processing units (GPGPU) is the recently-developed technique of using GPUs, processors specialized in graphics calculations, to perform general-purpose calculations. With the presence of programmable stages and precision arithmetic in the rendering pipelines, application developers are able to use stream processing on any data, not just graphics. The two largest GPU vendors, ATI and NVIDIA, started pursuing this market several years ago. NVIDIA began releasing cards supporting a C programming language API called Computer Unified Device Architecture (CUDA), allowing for a specially crafted C program to run on the GPU's stream processors. The GPUs have a parallel many-core architecture, where each core is capable of running thousands of threads simultaneously.

CUDA allows C programs to take advantage of the GPU's ability to perform calculations on large sets of data concurrently, while continuing to use the CPU for non-parallel calculations. It is also the first API that allows direct access to the GPU resources for general purpose usage without the limitations and complexities of using a conventional graphics API, such as

OpenGL or other direct 3D techniques. CUDA works with all GPUs from the G8X series onwards, including Geforce, Quadro and Tesla lines while maintaining binary compatibility between them. It allows some specific applications with extensive optimization and tuning to gain from impressive speedups, up to 647 times that of a CPU-only application. [8]. In general, the speedups are usually around the 10 times mark.

CUDA applications must be compiled with a specialized compiler created by NVIDIA, which compiles the C code into an intermediate assembly code called Parallel Thread Execution (PTX), which is then further compiled into the *Cubin* binary format. PTX is a low-level parallel thread execution virtual machine and instruction set architecture (ISA) [9].

The PTX assembly language has been fully documented by NVIDIA and made public, and has been chosen by OpenCL as the standard byte code to be generated not just by CUDA but by all OpenCL compilers. On the other hand, the Cubin binary format is proprietary and no information has been made public by NVIDIA, although the format has been partially documented by some developers by using reverse engineering [10]. Furthermore, only the most basic elements of the underlying hardware architecture have been documented by the vendor, and there are apparently no plans to do so in the future.

The closed nature of the Cubin format does not allow for any extensions or simulations. In this context, a simulator is a software application that models the architecture of a GPU, allowing the evaluation of different hardware designs without physically building them and obtaining detailed performance metrics. Simulations are required in order to fully analyze and possibly enhance heterogeneous systems based on GPUs. This is the main motivation behind this thesis. To achieve this, an alternative to the Cubin binary format must be developed, and since it will not be able to run on GPU hardware, an emulator of the GPU architecture must be developed. An emulator is defined as a system that duplicates the functionality of a system, making the latter behave like the former. The main objective of this thesis is to create an emulation library that parses the PTX intermediate assembly language generated from a CUDA application, generates a new binary object that substitutes Cubin and then uses it to emulate the functionality of a GPU and obtain the same numerical result.

1.2 Emulation Library Overview

The emulation library consists of several subcomponents.

- **PTX Parser**
  Parses the PTX intermediate assembly code generated by the CUDA compiler with a Flex/Bison lexical analyzer.

- **Intermediate Code Analysis**
  Performs Basic Block (BBL) identification on the parsed PTX instructions to generate a Control Flow Graph (CFG), on which it performs liveness analysis and interference analysis, allowing it to perform register allocation using Greedy Colouring, as the PTX code utilizes an unlimited number of registers.

- **OPBF Generation**
  The Open PTX Binary Format (OPBF) is the proposed alternative to the proprietary Cubin binary format by NVIDIA. An OPBF file is generated from the previously parsed PTX instruction in several passes to allow label resolution, since all labels are resolved to addresses. The OPBF also includes including a partial memory map of the kernel.

- **Emulation Library**
  Loads the OPBF file and emulates the GPU architecture to run the kernels contained in them. It is the most important component, since it must emulate all of the PTX opcodes and the memory subsystems, and must behave exactly the same as the GPU, in order to obtain the same numerical results. The chosen emulation method is *interpretation*, although a code cache is built, drastically reducing the actual number of lookups needed to execute an instructions.

- **CUDA Library**
  The CUDA API includes hundreds of functions that allow programmers to transfer data to and from the GPU, call kernels, and many others. Since this emulation library replaces NVIDIA's *libcudart* (CUDA runtime library), referenced functions must be implemented to allow the application to execute properly and bind the application to the emulation library.

1.3 Thesis Overview

The thesis contents, besides this introduction, are divided into three separate sections.

- **Background**
  Contains background information on GPUs, describes the NVIDIA CUDA computing model and hardware, existing emulators and modern emulation techniques.

- **Design and Implementation**
  Contains a detailed description of each section of the emulation library as described in Section 1.2, including implementation decisions and details of each part.

- **Experimental Evaluation**
  In order to evaluate the library performance and its correctness, several tests from the Parboil Benchmark Suite, developed by the Impact Research Group at the University of Illinois, were run. The results are presented in this section, along with a small description of each benchmark and the particular GPU features showed in each of them. In order to identify bottlenecks in the library's code, profiling was applied to the library running each benchmark. These results are also presented here.

- **Conclusions and Future Work**
  Contains a general conclusion to the thesis, including the general knowledge acquired during the thesis development and proposed future modifications and optimizations to the emulation library code in order to increase its performance.

Chapter 2

Background

2.1 GPU History

GPUs were initially designed as specialized processors attached to a graphics card, dedicated to accelerate graphics operations. In 1991, S3 Graphics developed the first single-chip 2D accelerator, the S3 86C911. By 1995, all major graphics card vendors had added 2D acceleration support. At that time, there was increasing demand for hardware-accelerated 3D graphics. Rendition Vérité V1000 was the first chipset that successfully integrated 2D and 3D functionality.

In 1999, NVIDIA released the GeForce 256, a revolutionary chipset that expanded the rendering pipeline to include a hardware transformation and lightning engine[1], revolutionizing the graphics card industry. This release established NVIDIA as the industry leading and marked the start of a steady decline that lead to the bankruptcy of numerous competitors, such as S3, Matrox and specially 3dfx. Only ATI was capable of directly competing with NVIDIA, releasing their Radeon series only a year after.

NVIDIA was the first vendor to produce a programmable shading capable chip in the GeForce 3 chipset in 2001. In 2002, ATI released the ATI Radeon 9700, where pixel and vertex shaders[2] could implement looping and floating point operations. Since then, NVIDIA and ATI have been the two major GPU vendors besides Intel, which is the current market leader with low-cost, less powerful integrated GPU solutions [11]. They have been alternatively taking the performance lead with different products. It should be noted that there are three minor vendors: VIA/S3 and SiS, with a 1% market share each, and Matrox with a 0.1% market share.

Since 2003, GPUs have steadily been leading the single precision floating-

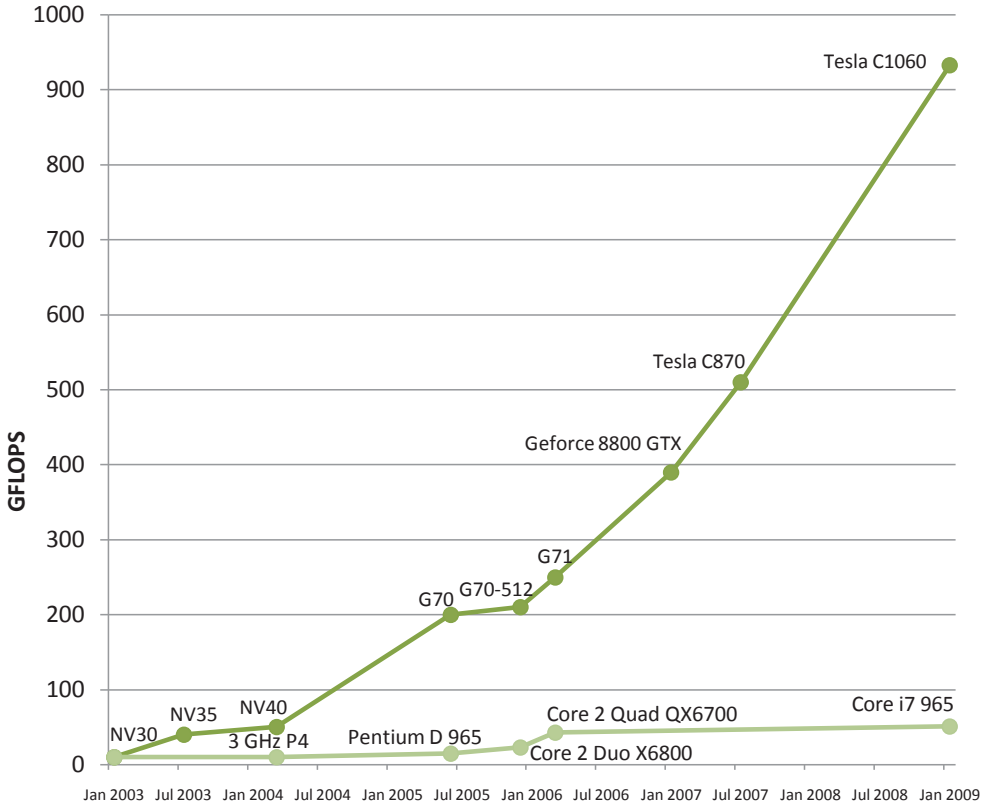

Figure 2.1: *GPU vs CPU Performance Gap* *(Based on [12, c.1, p.2])*

point performance race. CPUs, on the other hand, have slowed down their performance improvement significantly [12]. This can be clearly seen in Figure 2.1. It should be noted that the graph shows the theoretical peak speed that both CPUs and GPUs can achieve, which is not necessarily reachable. This performance gap is probably the reason for the current GPGPU trend and has motivated many application developers to move the computationally intensive parts of their applications to the GPU when tools have been made available.

As mentioned in Section 1.1, NVIDIA saw the potential for the GPU to use its huge computing power for uses outside of purely graphics calculations. In February 2007, they launched CUDA, a SDK and API that allows the development of applications that run on the GPU. It has proven to be very popular among the research community, as it can offer large performance benefits [12].

After being bought by AMD, ATI launched a similar SDK for their own cards called Stream SDK, formerly called Close to Metal (CTM). Even though the first general purpose computing efforts on a GPU was on AMD

ATI cards in October 2006 [13], NVIDIA strongly backed and publicized CUDA, leaving AMD ATI out of the public spotlight. Some people have claimed that the original CTM API was difficult to work with, besides coming much later than CUDA. Soon after releasing the Steam SDK which supposedly fixed the shortcomings of the CTM API, AMD ATI and NVIDIA announced future support for OpenCL [14].

2.2 The NVIDIA CUDA Architecture

There are three main abstractions in the CUDA architecture: thread groups, shared memories and barrier synchronization. These are all available to the programmer as a minimal set of C extensions. These abstractions allow programmers to partition a problem into sub-problems that can be solved independently in parallel. The problem can then be further split into smaller pieces that can be solved cooperatively in parallel. This approach can be applied to algorithms that can be expressed as data-parallel computations with a high ratio of arithmetic to memory operations.

2.2.1 Computing Model

The CUDA API allows programmers to define C functions which are known as *kernels*. The control-intensive parts of the algorithms are implemented in the host code, while the data parallel parts are implement in kernels. The host code is executed by the CPU, while the kernels are executed by the GPU. When called, kernels are executed in parallel N times by N different threads. Each of these threads is given a unique *thread* ID that is accessible to the programmer by the `threadIdx` variable. This allows to distinguish themselves from each other and identify the appropriate portion of the data to process, as they all receive the same parameters.

The host code and the device code (kernels) can be stored in the same source file. As we can see in Listing 2.1, both the kernel and host code are written in an extended version of ANSI C. The `__global__` keyword is used to declare a kernel, and the *`kernel_name`*`<<<X, Y>>>` syntax is used to specify the number of threads for each kernel call. This example adds two vectors A and B of size N and stores the result into vector C. Each of the threads that execute the kernel does a single addition, corresponding to its `threadIdx` value.

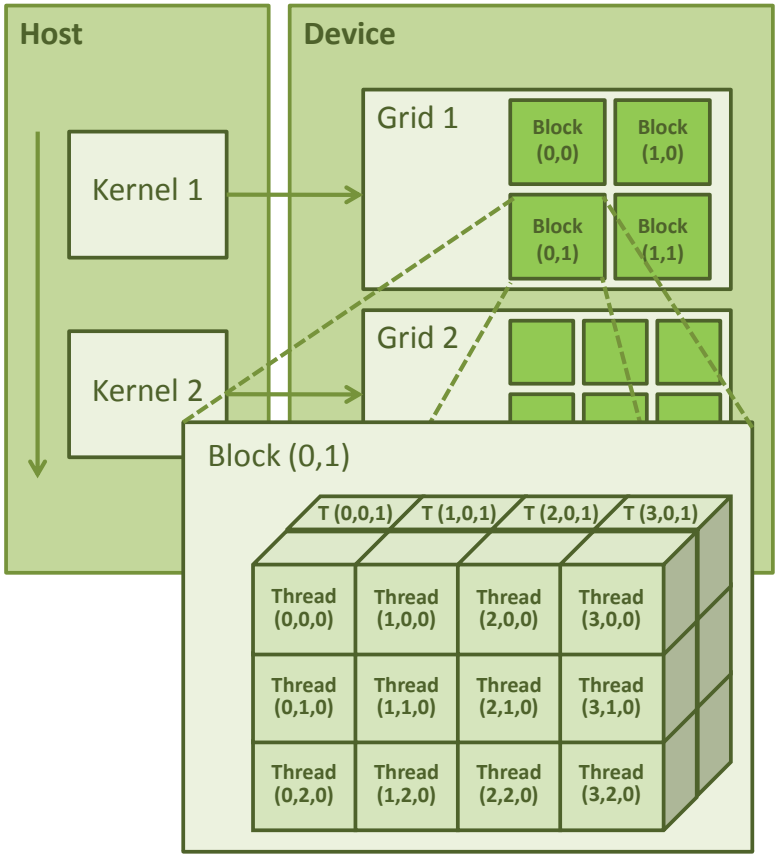

Figure 2.2: *CUDA Thread Hierarchy* *(Based on [12, c.2, p.10])*

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

int main(int argc, char *argv[])
{
    vecAdd<<<1, N>>>(A, B, C);
    return 0;
}
```

Listing 2.1: *Basic Kernel Example* *(Taken from [15, p. 13])*

CUDA organizes threads in a two-level hierarchy: *grids* and *thread blocks*. This can be seen in Figure 2.2. When a kernel is launched, it is executed by a grid of threads. This grid can be seen as a 2D array of thread

blocks. Each of the grids is usually formed by thousands to millions of GPU threads per kernel invocation. All blocks in a grid have the same number of threads, and are organized into a 3D array of threads. Each thread block has a unique 2D coordinate which can be accessed by the `blockIdx.x` and `blockIdx.y` variables. The number of blocks per grid can be accessed by the `gridDim.x` and `gridDim.y` variables. In the case of Listing 2.1, most of the hierarchy is ignored, and only one dimension of `threadIdx` is used. Since it is actually a 3-component vector with x, y and z coordinates, it allows threads to be identified using 1D, 2D or 3D indexes, forming 1D, 2D or 3D thread blocks.

The first parameter passed to the kernel is the number of blocks per grid. In Listing 2.1, this value was . The block dimensions (number of threads in each dimension of the block) is supplied by the programmer as the second parameter given at the kernel launch. In Listing 2.1, the block dimension was `N`. Since the NVIDIA Tesla architecture limits the maximum number of threads per block to 512, this is the total number of elements including all dimensions. The developer has flexibility to chose any distribution of these elements on each dimension, as long as the total number does not exceed 512. A kernel launch, is therefore defined as *`kernel_name`*`<<blocks_per_grid, threads_per_block>>`. It should be noted that once a kernel is launched, its dimensions cannot change during that invocation. From a programmer point of view, this is a disadvantage, since dynamic dimensions would allow kernel algorithms to be refined. From a performance point of view, it is an advantage, since fixed dimensions allow the hardware to not have resizing capabilities, therefore increasing performance. From our emulation point of view, supporting dimension resizing would complicate the implementation, and therefore the lack of it is an advantage. There is ongoing research[16] about dynamically resized kernels.

Threads within the same thread block can cooperate among themselves by sharing data. The sharing is done through the shared memory area and synchronization is performed by using the function `__syncthreads()`. This function performs barrier synchronization, a popular method of coordinating parallel execution. When called, all threads in a block will be stopped at the location until everyone else reaches it, ensuring that all of them have completed a phase of the execution before being allowed to continue. It should be noted that threads in different blocks do not perform barrier synchronization with each other, and therefore, different blocks can be executed in any order and in parallel or in series. This allows application scalability, as newer GPU cards support more blocks to be executed in parallel and

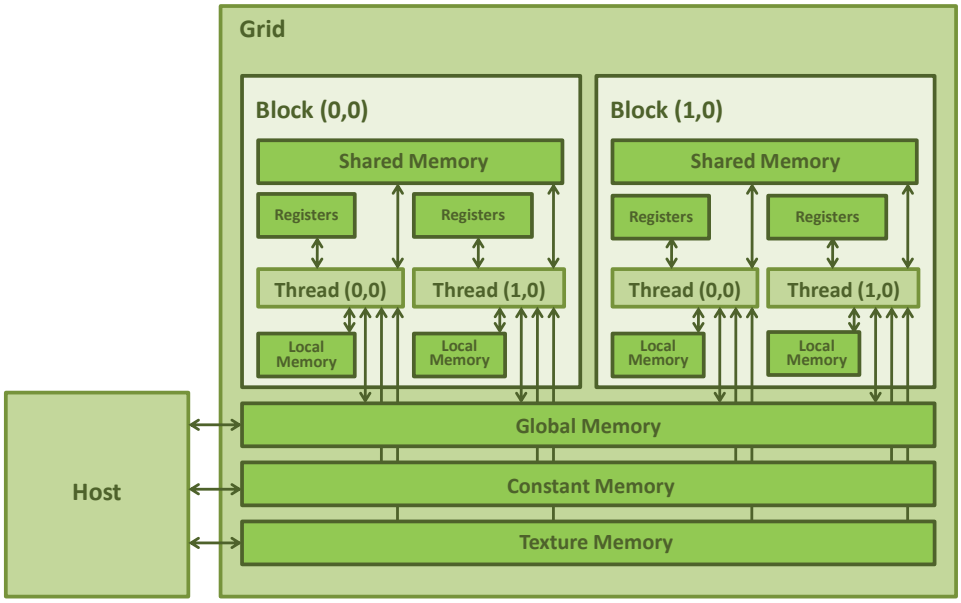

Figure 2.3: *CUDA Memory Hierarchy* *(Based on [12, c.4, p.3])*

therefore can benefit from the independence among different blocks.

There are several memory spaces which CUDA threads may access, as seen in Figure 2.3. Each thread has a specific local memory and registers, each thread block has a common shared memory and all threads have access to a global memory space. Additionally, the constant and texture memory spaces provide read-only access. Developers can declare variables into the various spaces by using specific keywords, such as `__device__ __shared__` or `__device__ __constant__`. Physical characteristics of the spaces are discussed in Section 2.2.2.

2.2.2 Computing Hardware

The NVIDIA Tesla architecture is based on a scalable array of multithreaded Streaming Multiprocessors (SMs). [15] A multiprocessor contains 8 Scalar Processor (SP) cores. The multiprocessor creates and executes threads concurrently in hardware with a virtually zero scheduling overhead. Fast barrier synchronization with the `__syncthreads()` function along with the zero-overhead scheduling allows for fine-grained parallelism: assigning, for example, one thread per pixel.

The multiprocessor employs an architecture called Single-Instruction, Multiple-Thread (SIMT). Each thread is mapped to one SP core, and each SP core executes independently with a dedicated instruction address and

register state. This technique is similar to Single Instruction, Multiple Data (SIMD), used in many architectures, such as the IBM Cell.

The use of the SM and SP terminology can lead to confusion. The behavior of an SM is closer to a core, as it performs fetch, decode and control on an instruction. An SP, however, is similar to an Arithmetic Logic Unit (ALU), as it just performs the instruction execution. The terminology is used by NVIDIA in its CUDA and PTX documentation, so we will adhere to it.

The multiprocessor manages threads in groups of parallel threads called *warps*. The size of this warps is implementation specific, in the GeForce 8800GTX, warps are made up of 32 threads. It should be noted that all threads in a warp start at the same program address, but they are obviously free to differ once branching occurs. Every time a new instruction can be executed, a warp that is ready to execute is selected and run. If threads on a warp diverge, the warp executes serially each path taken, a phenomenon called *branch divergence*. It only occurs in a single warp, as different warps will execute independently.

Warps play a crucial role in preventing long delay operations such as global memory access from slowing down the execution. When an instruction executed by a warp is waiting for the result of a long operation, the warp is placed on hold, and another warp that is ready for execution is loaded in its place. When the first warp is ready, it will be queued for execution. If there are enough warps, there will most likely always be one of them ready for execution, effectively delaying the execution but increasing throughput. This technique is called Simultaneous Multi Threading (SMT) [17], and tries to exploit parallelism across multiple threads, due to the fact that a single threads has a limited amount of ILP.

The GeForce 8800GTX, for example, has 16 SMs, allowing up to 128 thread blocks to be executed simultaneously. One important SM limitation is the number of threads that can be scheduled. In the GeForce 8800GTX, the limit is 768 threads per SM (24 warps), which would result in a total of 12288 threads simultaneously executing.

Figure 2.4 shows how the CUDA memory hierarchy is implemented in the SIMT architecture. Every processor core has a complete set of 32-bit processors, a shared memory space, shared by all cores in a SM and read-only constant and texture cache memories shared by all cores.

In the GeForce 8800 GTX implementation, each SM has 8000 registers. Since each SM can hold up to 768 threads, this amounts to a total of only 10 registers per thread. If each thread uses more than 10 registers, the total

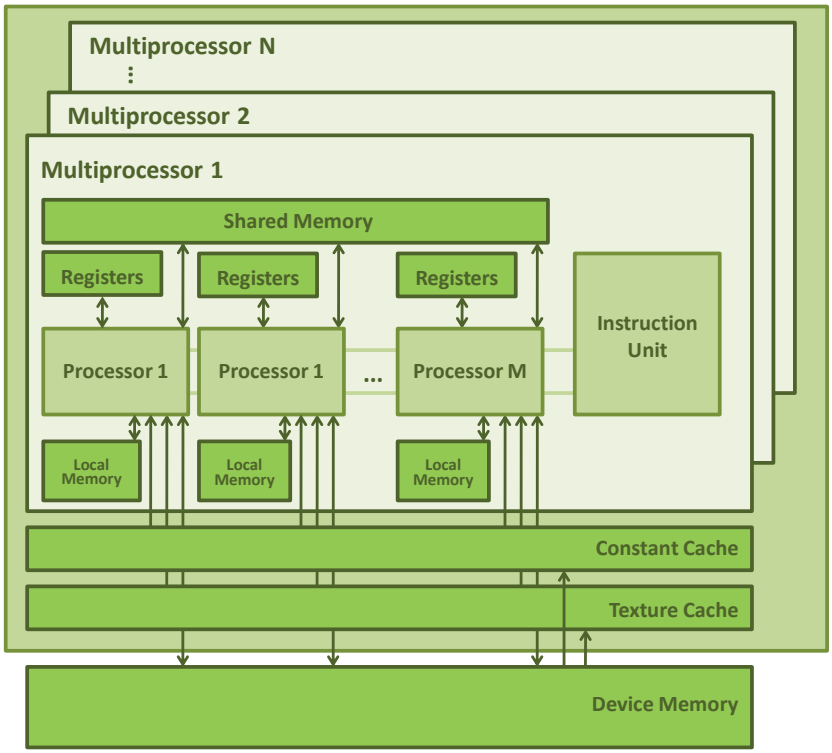

Figure 2.4: *GPU Hardware Model* *(Based on [15, p.14])*

number of threads will be reduced. Since this penalty occurs at the block level, the number of threads will be reduced significantly, such as 128 if each block contains 128 threads. This is a clear example of the fact that memory is a limiting factor in CUDA parallelization, and so code must be optimized to utilize the least amount of available resources. The number of threads can reduce the number of warps available, and therefore increase the latency when many long-latency operations such as global memory accesses occur. Besides the register limiting factor, excessive shared memory usage can lead to the same problem. In the GeForce 8800 GTX, there are 16kB of shared memory in each SM, so each block should not use more than 2kB of it, or the number of blocks will be reduced as described before.

Memory characteristics of the different spaces available in the CUDA architecture as implemented on a GeForce 880GTX GPU are shown in Table 2.1. The global memory serves as the storage area for the texture and constant areas. Access to global memory is more efficient when multiple threads access contiguous elements. On the other hand, shared memory, whose scope is limited to threads in the same thread block, is used as a local scratchpad. Constant memory is often used for lookup tables, due to its low

Space	Size	Latency	Read-Only
Global	768MB total	200-300 cycles	RW
Shared	16kB/SM	≃ Register Latency	RW
Constant	64kB total	≃ Register Latency	RO
Texture	Up to global	> 100 cycles	RO
Local	Up to global	200-300 cycles	RW

Table 2.1: *Memory spaces in a GeForce 880GTX GPU (Based on [8, p.3])*

latency. Texture memory can be used to perform hardware interpolation and can be configured to wrap around the edges of the texture. These characteristics are used in certain applications such as video encoding. Local memory is usually used for register spilling.

2.3 Emulation

2.3.1 Techniques

An emulator duplicates the functionality of one system using a different system, in a way that allows the second system to behave exactly like the first one. [18] The following is a brief list of currently employed emulation techniques.

- *Binary translation*[19, 18] is the emulation of a CPU instruction set (*source*) on a different instruction set (*target*) by using code translation. In *static* binary translation, a complete executable file for the source architecture is translated into an executable file of the target architecture. On the other hand, *dynamic* binary translation processes a short sequence of code, typically a single basic block, translating it and caching the result.

- *Native execution*[20], also called *direct execution*, is the execution of instructions of an emulated CPU directly on the host CPU. This is obviously only possible when both instruction sets are the same. Privileged instructions such as I/O are usually not directly executed, as they would affect the operating system running the emulation. Instead, they are usually interpreted.

- *Dynamic recompilation*[21, 18] is a feature utilized by some emulators where the emulator recompiles part of a program during execution.

This is usually done to produce more efficient code that perhaps was not available to the compiler or perhaps due to a poor optimization. It is also known as *just in time* compilation, or JIT.

- *Interpretation*[18] is the slowest emulation technique available, usually performed by parsing the emulated assembly code in a high-level language such as C, and then performing the operations described in the code.

- *Virtualization*[20] is a technique used to implement a virtual machine environment, completely emulating the underlying hardware. All software that can run on the simulated hardware should run on the virtualized environment without any modification. A key aspect of virtualization is the emulation of I/O and other privileged operations that can not be executed directly and must be emulated. It requires instruction set (ISA) compatibility between the emulated and emulating hardware.

2.3.2 Emulators

The following is a list of several modern emulators which utilize the techniques described in Section 2.3.1 and which were taken into consideration when designing the PTX emulator developed in this thesis.

- *QEMU*[22] is a generic open source machine emulator and virtualizer. As a processor emulator, it allows running operating systems and applications made for one architecture on another one. Besides CPU emulation, it also provides a set of device models for peripherals such as video, network or sound cards, which allow it to run numerous operating systems without any modification.

  The host CPUs supported are x86 (both 32-bit and 64-bit) and PowerPC, while the main supported target emulated CPUs are x86, ARM, SPARC, MIPS, m68k and CRIS. It uses dynamic translation to achieve reasonable performance, and an accelerated mode supporting a mixture of binary translation and native execution.

  Several software-based accelerators were written to speed up execution. The main one was KQEMU, which speeds up x86 on x86 emulation, by running user mode code directly on the host CPU instead of emulating it. Unlike Linux Kernel-Based Virtual Machine (KVM), it does not require hardware CPU virtualization. Linux KVM is a

virtualization infrastructure implemented in the Linux kernel which supports native virtualization using Intel VT or AMD-V hardware virtualization support.

- *Bochs*[23] is a cross-platform open source x86 (32-bit and 64-bit) emulator. It supports emulation of several processors (386, 486, Pentium up to Pentium 4, and several 64-bit processors) and peripherals, allowing the execution of many guest operating systems such as a Linux, DOS or Windows. Since it uses interpretation and is written in cross-platform code, it runs on many non-x86 systems such as Solaris (Sparc), GNU/Linux (PowerPC/Alpha), MacOS (PowerPC), IRIX (MIPS), BeOS (PowerPC), Digital Unix (Alpha) and AIX (PowerPC). Its emulation method results in very slow performance, and so it is mainly used by hobbyists or OS developers.

- *PearPC*[24] is a cross-platform open source PowerPC emulator, allowing the execution of operating systems such as a PPC Linux or Mac OS X to run in other architectures, mainly x86. Besides the CPU, it also emulates a PCI bridge, an IDE controller, a network controller, among others. If the host is running in an x86 processor, it translates PowerPC instructions into x86 instructions and caches the results, incurring in a slowdown of only 15 times than the host. On non-x86 platforms, it fully emulates the CPU, making it run 500 times slower than the host.

- *VMWare Inc.*[25] is the major virtualization software vendor. Its products provide virtualized peripherals and processors, allowing users to set up multiple x86 (32-bit and 64-bit) virtualized operating systems. They do not emulate an instruction set for different hardware not physically present, utilizing native execution on user code and dynamic translation for kernel code, providing a significant performance boost, running at 80% of the achievable speed.

- *VirtualBox*[26] is an x86 virtualization software, originally created by Innotek and currently developed by Sun Microsystems. It is very similar in features and characteristics to *VMWare Workstation*, but besides a commercial version, an Open Source edition is available. It supports hardware virtualization using the Intel VT-X and AMD AMD-V extensions.

- *MINT*[27] (MIPS Interpreter) is a simulator for multiprocessor systems developed in 1994. It provides a set of simulated processors that run executables compiled for a MIPS R3000 multiprocessor, producing a list of memory events that eases the development of memory simulators. It uses a hybrid technique of software interpretation and native execution, minimizing the emulation overhead.

- The NVIDIA *deviceemu* (device emulation) emulates a G80 GPU, allowing CUDA applications to run directly on the CPU. It is activated by appending the compile option `-deviceemu` to nvcc while compiling the application. The emulation technique utilized is unknown, as the underlying library is closed source and not documented by NVIDIA.

2.4 Software Profiling

Software profiling is the investigation of program behavior using information collected during program execution, as opposed to by analyzing the source code prior to execution [28]. It allows performance analysis, by determining which functions or sections of an application could benefit from optimization. There are two types of software profiles. *Statistical profilers* operate by sampling the program counter of the application at regular intervals, by using interrupts. They allow the target program to run almost at full speed, although the results presented are not accurate. *Instrumenting profilers*, while slowing down the application several orders of magnitude, provides accurate results.

The most commonly used open source statistical profilers are *gprof* and *oprofile*. *Gprof* is only capable of analyzing processes in *user* space for a given process, so external shared libraries and the OS kernel can not be checked for bottlenecks. *Oprofile*, on the other hand, allows shared libraries and the OS kernel to be checked.

The most commonly used instrumental profiler is *valgrind* [29]. The original program does not run directly on the processor. Instead, by using dynamic recompilation, its code is translated into an intermediate representation, which is a processor-neutral, SSA-based form. After the conversion, a *valgrind tool* can alter the intermediate code, before *valgrind* converts it back to machine code to let the host processor run it. This causes execution be slowed down at least 4 to 5 times. There are multiple tools available, that allow checking for memory leaks and other memory related problems, generating callgraphs or detect race conditions.

Chapter 3

Design and Implementation

As explained in Section 1.1, the main objective of this thesis is to create an emulation library that parses the PTX intermediate assembly language generated from a CUDA application, generates a new binary object that substitutes Cubin and then uses it to emulate the functionality of a GPU and obtain the same numerical result. A general work flow graph for the emulation library is shown on Figure 3.1. The proposed and implemented emulation scheme has two main parts: compilation and execution.

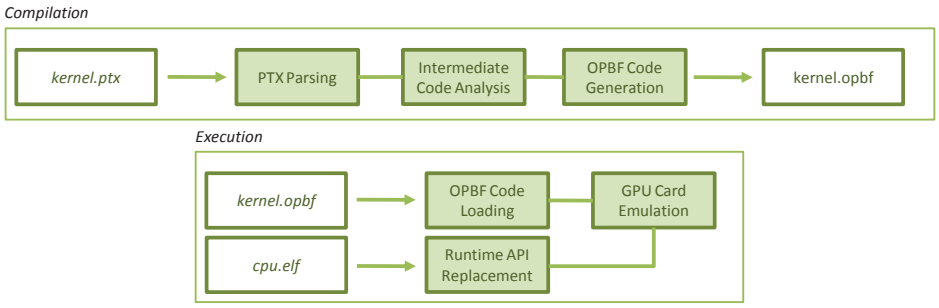

Figure 3.1: *General Emulator Workflow*

In the compilation part, we parse the PTX intermediate assembly code with a lexical analyzer, loading the assembly code into an organized data structure. This allows the intermediate code to be analyzed, which includes generating a control flow graph, performing liveness and interference analysis, and, finally, performing register allocation. Then, a binary OPBF file, which substitutes the proprietary Cubin format, is written. It contains the parsed and analyzed intermediate code. In the execution part, the previously generated OPBF file is loaded, allowing the stored instructions to be emulated.

Block	Library	Status
PTX Parsing	*libptx*	*Minor additions and bug fixes*
Intermediate Code Analysis	*libasm*	*Major modifications*
OPBF Code Generation, Loading	*libopbf*	*Developed from scratch*
Runtime API Replacement	*libecuda*	*Developed from scratch*
GPU Card Emulation	*libemul*	*Developed from scratch*

Table 3.1: *Emulation Library Parts*

Table 3.1 shows which library performs each of the blocks in Figure 3.1, and whether each block was modified or developed from scratch. The PTX parsing library, *libptx*, was already implemented and working, and only minor modifications and bug fixes had to be applied. The intermediate code analysis library, *libasm*, was also previously implemented, but major parts had to be rewritten, such as the register allocation algorithm. The OPBF code generation library, *libopbf*, the runtime API replacement, *libecuda*, and the GPU Card Emulation, *libemul*, were developed from scratch.

The language of choice for all of the libraries comprising this project is *C++*. It provides an excellent combination of high level and low level features while delivering excellent performance. It was chosen over *C* because the size and complexity of the project benefit enormously from using *Object Oriented Programming*. It has also provided a key feature of the emulation process, *templates*, which will be described in Section 3.5.

There are currently several projects underway that are trying to achieve the same objectives as this thesis. *GPGPU-Sim* [30] was developed by researchers at the University of British Columbia. The source code of the tool was published in July 2009, but is only available to approved members of a private *Google Groups* group, so the internal workings are unknown. Barra [31] was developed by researchers at the Université de Perpignan. Instead of parsing the PTX output, Barra uses reverse-engineering techniques to decompile the binary Cubin file. Finally, Ocelot [32] was developed by researchers at the Georgia Institute of Technology. Source code was released on a BSD License on July 2009. A brief examination of the project revels that it is very similar to this thesis, but strong emphasis has been put on supporting all of the NVIDIA CUDA SDK sample applications, leaving out important parts such as Register Allocation. It is the most complete emulator available at the moment.

3.1 Overview

nvcc is the NVIDIA-provided compiler that simplifies the process of compiling CUDA applications [33]. It invokes a collection of tools that implement the different compilation stages. A simplified flowchart of the compilation process is shown on Figure 3.2. The first part, performed by the *cudafe* tool, separates host (CPU) code from device (GPU) code. The device code is then compiled into the propietary Cubin format from the generated PTX intermediate assembly code. The device code can be loaded from the external Cubin file using CUDA API calls or optionally be incorporated into the host device code, as a global initialized data array. In the latter case, the host code contains extra code included by nvcc, including the translation code needed to load and launch the kernel. As noted in Figure 3.2, the compilation to the Cubin format is propietary.

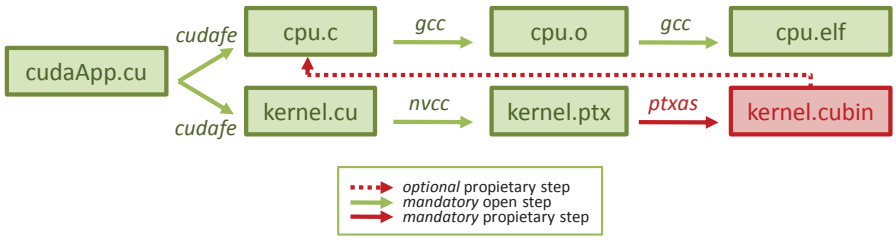

Figure 3.2: *Simplified NVIDIA CUDA compilation flow*

Since the Cubin format is proprietary, undocumented, and subject to change, it cannot be used as the basis of our emulator. The step prior to the Cubin assembly is the generated intermediate assembly language PTX code. This code will be parsed and compiled into a Cubin format substitute, named OPBF (Open PTX Binary Format). The modified compilation flowchart is shown in Figure 3.3. The original workflow still applies, although the option to include the resulting Cubin data into the host code is now mandatory. Although the Cubin data itself will not be used, the translation and launching code included in the host code is needed.

In order to create a Cubin alternative, the PTX intermediate assembly code must be parsed and analyzed. No code optimizations are applied on the parsed code, but register allocation through liveness analysis is performed to reduce the number of registers to a reasonable number. The resulting code is then encapsulated in a binary OPBF file. This is the last step of the compilation process.

When a CUDA application with embedded Cubin data is started, a ker-

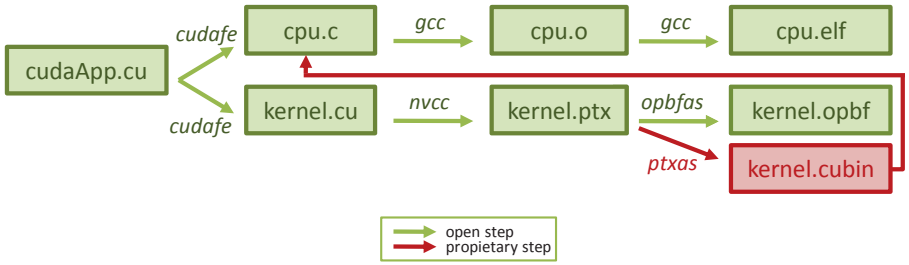

Figure 3.3: *Modified libecuda compilation flow*

nel call such as `matrixMul<<< grid, threads >>>(d_C, d_A, d_B, WA, WB);` in the host code is translated into a normal C function call, such as `__device_stub__Z9matrixMulPfS_S_ii(d_C, d_A, d_B, 3*16, 8*16)`, preceded by a call to the API function `cudaConFigureCall( grid, threads)`. The *device stub kernel* function is defined in the kernel cpp file generated by nvcc, and is simply a wrapper function that contains calls to the API functions such as `cudaSetupArgument()` or `cudaLaunch()`, which configures and launches a kernel, respectively.

All of the CUDA API functions are provided by the *libcudart* library, the CUDA runtime library provided by NVIDIA. To allow emulation, a drop-in replacement of the runtime library, with a re-implementation of all the functions, will be developed. A call to the `cudaLaunch()` function, for example, will not load and run the Cubin data in the GPU as the original function probably does. Instead, it would instruct the emulation library to load and run the previously generated OPBF file inside the emulated environment.

3.2 PTX Parser (libptx)

In order to analyze the code in the PTX intermediate assembly language produced by the nvcc compiler from the higher level kernel source code, it must first be parsed. A PTX file contains plain-text assembly language instructions, following the instruction set specification available in [9]. Listing 3.1 contains a small fragment of the PTX code generated from the matrix multiplication code.

Parsing complex strings that follow a formal description such as PTX code is usually done by using a parser generator. Its input is the grammar of a programming language, while its output is the source code of a parser. Basically, it eases the otherwise complex task of creating a parser

word	token
mul24	INSTRUCTION
lo	BITS
s32	TYPE
$r2	REGISTER
,	COMMA
$r1	REGISTER
,	COMMA
16	INTEGER

Table 3.2: *Sample tokenization of the PTX code* `mul24.lo.s32 $r2,$r1,16`

by hand, which would usually involve a large number of string comparisons and conditional code. The tools chosen for the task were the flex/bison duo [34].

```
77 cvt.s32.u16 $r1, %ctaid.x;
78 mul24.lo.s32 $r2, $r1, 16;
79 cvt.s32.u16 $r3, %ctaid.y;
80 ld.param.s32 $r4, [__cudaparm__Z9matrixMulPfS_S_ii_wA];
81 mul.lo.s32 $r5, $r3, $r4;
82 mul.lo.s32 $r6, $r5, 16;
83 add.s32 $r7, $r6, $r4;
84 sub.s32 $r8, $r7, 1;
85 cvt.s32.u16 $r9, %tid.x;
86 cvt.s32.u16 $r10, %tid.y;
87 ld.param.s32 $r11, [__cudaparm__Z9matrixMulPfS_S_ii_wB];
88 setp.lt.s32 $p1, $r8, $r6;
89 mov.f32 $f1, 0f00000000;
90 @$p1 bra $Lt_0_17;
```

Listing 3.1: *matMul PTX code fragment*

Flex (fast lexical analyzer generator) is a free implementation of lex, an automatic lexical analyzer, allowing the conversion of a character sequence into a token sequence. It requires a set of rules, specifying which set of characters form each token. Take, for example, line 78 of Listing 3.1. It could be tokenized as shown in Table 3.2. The grammar shown should be completely defined in a file which is passed to flex. All possible tokens should be defined in that grammar file. Lex would then output that grammar, serving as the input for bison. GNU bison is a free implementation of yacc,

which is a parser generator, that converts a grammar into C parser code for that grammar.

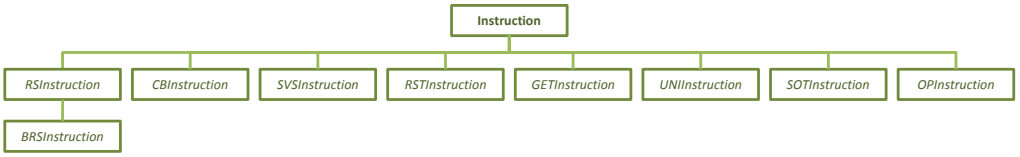

Figure 3.4: *`Instruction` Classes Hierarchy*

Parsed instructions are stored in a hierarchy of C++ classes, which all derive from the base `Instruction` class. These classes, as shown in Figure 3.4, are `RSInstruction`, `BRSInstruction`, `CBInstruction`, `SVSInstruction`, `RSTInstruction`, `GET Instruction`, `UNI Instruction`, `SOTInstruction` and `OPInstruction`. Instructions are organized into these different types according to the number of modifiers and operands. The *MUL24* opcode performs the multiplication of the 24 most significant bits of a register and an integer, and stores the result in a third register. It can have up to three modifiers: bits to consider, rounding and saturation. Therefore, it is a *BRS* instruction. An `Instruction` object contains all the instruction information, such as the operands, modifiers or source and/or destination type.

Obviously, a proper flex grammar definition must be in place in order for the bison code to be straight forward. Listing 3.2 shows a bison semantics snippet for the previously mentioned *MUL24* opcode. It shows how the opcode is parsed as a sequence of tokens: the *MUL24* token, a *bit* token, a *TYPE* token, and so on. Then, the `Instruction` object is created and initialized with the instruction modifiers by the `BRSIns` function. The `addReg` and `addValue` store the instruction parameters. In this case, the destination and source registers and the multiplication constant.

```
| MUL24 bit TYPE reg COMMA reg COMMA value {
    BRSIns(_mul24, (bit_t)$<i>2, _no_rnd, false, (type_t)$<i>3);
    addReg(Dst, (type_t)$<i>3, $<reg>4);
    addReg(Src, (type_t)$<i>3, $<reg>6);
    addValue(Src, (type_t)$<i>3, $<value>8);
}
```

Listing 3.2: *`MUL24` opcode bison semantics*

A basic working version of the code in libptx, including the PTX grammar and parser, had already been developed by my advisor Isaac Gelado

prior to the beginning of this thesis. There have been, however, numerous additions and bug fixes applied to the code during the thesis development, including the following.

- Added the original PTX line number to an instruction to ease the debugging process, particularly basic block visual identification, which can now be done by the start and ending source code line.

- Added support for zero-sized vectors, which use all the available shared memory space. This technique is used by the *sad* benchmark. (Section 4.1)

- Although PTX defines all fundamental types (signed integer, unsigned integer, floating point, untyped) with different lengths (8, 16, 32 and 64 bits), these lengths were originally simplified to just 32 and 64 bits. This simplification worked for most cases, but it did not for the *sad* benchmark, as several kernel parameters were 16 bit integer arrays, which were incorrectly treated as 32 bit integer arrays due to the simplification. Because of this, full support for all lengths was added. More details are available in Section 3.4.

- Added support for vectors in the *LD* and `ST` opcodes, which allow direct access to up to four-element vectors in a single instruction. These instructions are parsed as a single instruction, but they are split into separate instructions for the OPBF generation. Therefore, an instruction such as `ld.global.v4.u32 $r9,$r10,$r11,$r12, [$r8+0]` is split into four separate instructions: `ld.global.u32 $r9,[$r8+0]`, `ld.global.u32 $r10,[$r8+4]`, `ld.global.u32 $r11,[$r8+8]` and `ld.global.u32 $r12,[$r8+12]`.

- Added grammar support for the *TEX* opcode, as described in Section 3.5.

- Fixed a register allocation bug, as described in Section 3.3. It was caused by the *ST* opcode when used with indirections, such as in the example case `st.volatile.shared.u32 [$r107+0], $r114`. In this case, register 114 was marked as the source register and register 107 was marked as the destination register. This was incorrect, due to the fact that they are both source registers, as register 107's value is used to determine the actual memory address to access. This bug caused register 107 to be overwritten before reaching this instruction, as it

was marked as a destination and not as a source, and therefore was reused by the register allocation process.

- Fixed minor bugs, such as the incorrect parsing of the *MUL24* opcode, the incorrect parsing of negative float values.

- Added support for the `volatile` modifier in the grammar, which was added in the PTX ISA 1.1. Due to the emulation architecture, this modifier is simply ignored, as all memory locations are considered volatile.

3.3 Intermediate Code Analysis (libasm)

The parsed instructions that result from the parsing performed by *libptx* are stored in a vector of *Instruction* objects. *libasm*'s primary goal is to perform register allocation on the PTX code. As seen on Listing 3.1, the generated intermediate assembly language code does not perform register allocation: it always uses a new register, without reusing unused ones.

Register allocation is the process of assigning a big number of variables onto a reduced number of CPU registers. On a GPU architecture, the number of registers available is abnormally large (8000) compared to a conventional CPU (16 in a 32-bit x86 architecture). As noted in Section 2.2, however, if the number of registers per thread is greater than 10, the total available number of threads will be reduced. Therefore, register allocation is surely performed on the PTX code when assembling the Cubin format. Emulating it correctly, therefore, requires a reasonable register allocation to be performed on the code.

A basic requirement of register allocation is building a control flow graph (CFG) of the application code. A CFG represents all paths that can be taken during the execution of an application. Figure 3.6 shows a simple CFG. In a CFG, each node presents a basic block (BBL), which is a group of consecutive instructions without any jumps or any jump destinations. Therefore, a jump destination marks the start of a BBL, and a jump marks the end of a BBL.

In order to construct the CFG, all BBLs in the code must be identified. In our case, this is performed in two steps. First, when parsing the PTX code, as described in Section 3.2, a first approximation is to split the code into BBLs according to those lines with labels: these mark the start of at least some of the BBLs. Further along, in *libasm*, these BBLs are in-

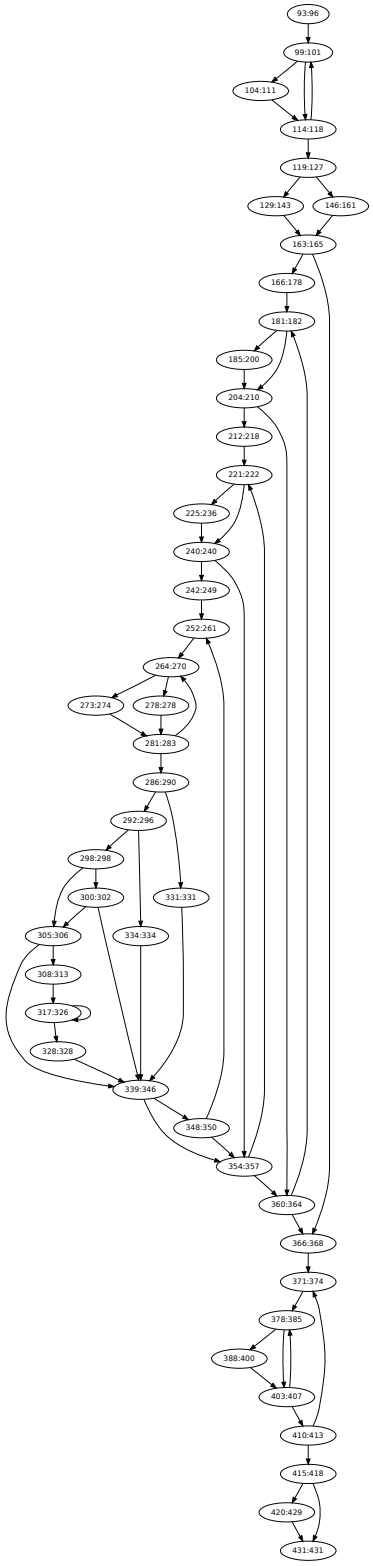

Figure 3.5: *tpacf benchmark CFG*

spected for jump instructions, and split accordingly, resulting in a complete identification of all BBLs.

An initial version of the register allocation implementation was developed by my advisor Isaac Gelado. Basically, registers were freed when a their values were not found to be used in the remaining BBLs of the program flow. Then, a register map of the allocated registers was propagated to each BBL edge. This version worked on small CFGs, such as the matrix multiplication benchmark (Section 4.1) CFG shown on Figure 3.6, and was relatively efficient, in that it allocated a small number of registers. The BBL node names identify the source code line intervals contained in each block. The graph nodes, BBLs, were visited in a *depth-first search* order, in the case of Figure 3.6, this order would be 77:90, 91:122, 126:188, 189:189, 194:203, 191:191.

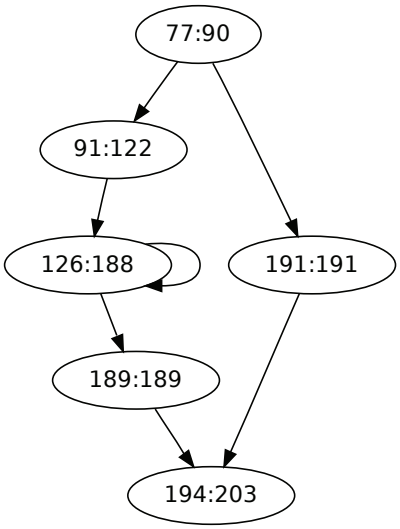

Figure 3.6: *matMul benchmark CFG*

Problems arose, however, when more complex benchmarks such as the *tpacf* benchmark were tested. The complete CFG for this benchmark is shown on Figure 3.5. As we can see, this was a much more complex graph, and after extensive debugging, the register allocation algorithm was found to be failing. Later on, it was discovered that although the PTX code was in static single assignment form (SSA) for small kernels, it was not for larger-sized kernels, leading to complications.

Figure 3.7 shows a small subset of the *tpacf* benchmark CFG, from its upper part. As we can see from the graph, the program execution flow could take two distinct paths, either through the 129:143 BBL (lines 129

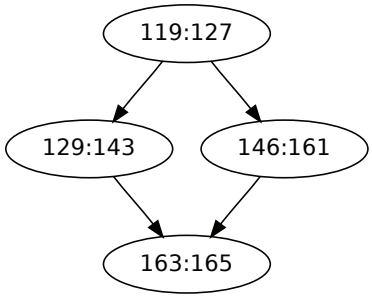

Figure 3.7: *Subset of the tpacf benchmark CFG*

to 143 from the PTX code) or thorough the 146:161 BBL. The register allocation algorithm failed to analyze that, even if a register was not used in the path that goes through the 129:143 BBL, it could be used in the path that went through path 146:161. This caused several registers to be freed and overwritten in the 119:127 BBL, even if they were used in the 146:161 BBL.

In order to fix this bug, register dependencies had to be propagated to parallel branches. That is, when reaching node 129:143, registers that were previously allocated in 119:127 should not be reused in node 129:143. The same reasoning applied to node 146:161. Then, when reaching a *collecting* node such as 163:165, registers that could be freed in both branches are actually freed. In order to perform this, the CFG must be visited in a different order, a *balanced search* in this case, which in this case would be 119:127, 129:143, 146:161 and finally 163:165.

This solution was successfully implemented, although it resulted in an excessively high number of registers, due to it being an excessive conservative approach, specially in very highly branching cases such as those shown in Figure 3.5. Problems arose, however, when applying the algorithm to the *rpes* CFG. Several hacks had to be applied in the algorithm to correctly traverse that CFG. It was clear that the existing register allocation approach was incorrect, with a high number of registers consumed and code changes needed when the complexity of the CFG increased, due to the more challenging *balanced* graph visiting.

Several popular register allocation algorithms were then considered, to completely replace the existing code. There are several long-established algorithms to perform register allocation, the most popular one being graph coloring. This method usually results in an effective allocation without a major cost in compilation speed [35]. The final working register allocation that was implemented was via coloring of chordal graphs [36, 37].

In this implementation, register allocation by graph coloring [35] is performed in the following steps, as shown in Figure 3.8. First, liveness analysis must be performed on the previously obtained CFG. Then, interference analysis is performed on the CFG, building an *interference graph*. Finally, registers are assigned by greedy coloring.

Figure 3.8: *Register Allocation Workflow*

Liveness analysis is performed by calculating, for each BBL in the CFG, which registers are *live* in each BBL of the CFG. A register is considered to be *live* if its value is used at some point during the remainder of the code, without redefining it. It is *dead* if its value is redefined before using it, or not used at all. We have implemented liveness analysis by using backward propagation [38]. Let us take, for example, the instruction `x = y + z`. According to this implementation, `y` and `z` are *live* at this point since they are used in a calculation, while `x` is *dead* since its value is overwritten. It is easy to go backwards, starting from the last instruction, and apply these two rules. This approach is only valid on straight-line code, without jumps and conditional branches. In that case, several iterations should be made until no further information is acquired, typically 2 or 3 iterations are needed.

Two registers are said to *interfere* if there are two BBLs where they are simultaneously live. In this case, it is possible that they contain different values at this point. Therefore, two registers that do not interfere can be merged into a single register. The interference graph is built to hold the interference information of all the registers. The nodes of the interference graph are the application registers. An *undirected edge* is inserted between two nodes if the two *virtual* registers interfere and should be assigned to different *real* registers. For a general instruction, an edge is created between the assigned register in that instruction and any future live registers. On a `mov` instruction, an edge is created between the destination register and all live registers live except for the source register.

Register allocation is performed via *graph coloring* [36, 37]. The problem of assigning colors to a graph (coloring) with K colors is NP-complete for $K \geqq 3$. Fortunately, programs in Static Single Assignment (SSA) form have *chordal graphs*, which makes the problem achievable in linear time. A program is said to be in SSA form if every variable is assigned exactly

once. Even if the program is not in strict SSA form and does not have a strict chordal graph, as some of our benchmarks, the algorithms behave reasonably well. In that case, the minimal number of registers (colors) is not used.

A node in a graph is *simplicial* if all neighbors of a node are connected to each other. An ordering of all nodes in a graph is called *simplicial elimination ordering* if the last node of the list is simplicial. A simplicial elimination ordering can be found in linear time by implementing a *maximum cardinality search*. Given an ordering, greedy coloring is applied by assigning colors to the vertices in the exact simplicial order that has been found, always using the lowest available color. If the graph is chordal, this algorithm is guaranteed to use the fewest possible colors [38].

The chosen register allocation implementation is simpler than other more efficient implementations, and works for all tested CFGs. It provides reasonable efficiency: the matrixMul kernel occupies 24 registers. The original implemented algorithm used 17 registers, while the modified algorithm used 24 registers. The NVIDIA implementation of the PTX compiler uses only 14 registers, mainly due to dead-code elimination.

3.4 OPBF Generation (libopbf/opbfas)

OPBF files are generated from a PTX file by the *opbfas* tool, which uses the previously described *libptx* and *libasm* to produce an OPBF file from a PTX file. Listing 3.3 shows a basic outline of the opbfas code. First, the input PTX file is parsed using libptx, storing a vector of `Instruction` objects, as described in Section 3.2. Then, register allocation is performed on the code, as described in Section 3.3. Finally, OPBF instructions are generated from PTX instructions and the binary file that contains the instructions is written.

```
open(PTX file);
parse(); /* using libptx */
registerAllocation(); /* using libasm */
generateBinaryCode(); /* using libopbf */
writeBinaryCode(); /* using libopbf */
```

Listing 3.3: *opbfas basic pseudo-code*

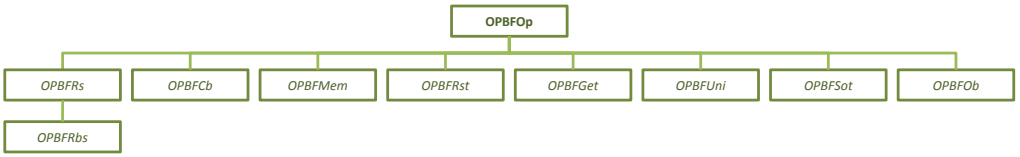

Figure 3.9: *OPBFOp Classes Hierarchy*

```
OPBFOp *OPBFOp::create(const PtxProgram &global, const PtxFunction
       &function, const Labels &labels, const Instruction &i)
{
   switch(getFormat(i.op())) {
      case _rbs:
         if(i.op() == _mul || i.op() == _div) {
            return new OPBFRbs(global, function, labels,
                  dynamic_cast<const BRSInstruction&>(i));
         } else {
            return new OPBFRs(global, function, labels,
                  dynamic_cast<const RSInstruction&>(i));
         }
      case _cb:
         return new OPBFCb(global, function, labels,
            dynamic_cast<const CBInstruction&>(i));
      case _svs:
         return new OPBFMem(global, function, labels,
            dynamic_cast<const SVSInstruction&>(i));
      case _rst:
         return new OPBFRst(global, function, labels,
            dynamic_cast<const RSTInstruction&>(i));
      case _get:
         return new OPBFGet(global, function, labels,
            dynamic_cast<const GETInstruction&>(i));
      case _uni:
         return new OPBFUni(global, function, labels,
            dynamic_cast<const UNIInstruction&>(i));
      case _op:
         return new OPBFOb(global, function, labels,
            dynamic_cast<const OPInstruction&>(i));
   }
   return NULL;
}
```

Listing 3.4: *OPBFOp creation from a PTX Instruction*

Similarly to the `Instruction` class hierarchy, `OPBFOp` is a class hierarchy, with `OPBFRs`, `OPBFRbs`, `OPBFCb`, `OPBFMem`, `OPBFRst`, `OPBFGet`, `OPBFUni` and `OPBFOb` as subclasses. This is shown in Figure 3.9. Each of these subclasses represents the same type of instructions that its `Instruction` class counterpart represents. An `OPBFOp` object has all the instruction attributes needed to perform an operation: the exact instruction type, a source and destination register vector, an immediate value vector, among others.

The OPBF instructions are generated from the PTX instructions in two passes, to be able to resolve all labels. This is due to the fact that, although PTX code uses labels for both jump destinations and to reference user variables (as seen in Listing 3.1, lines 90 and 80 respectively), in order to reduce the number of lookups performed during the emulation process, all labels are resolved to memory addresses at compilation time, since we are generating *static code*.

In the first pass, a new `OPBFOp` object is generated for each instruction. The `OPBFOp::create` method, shown in Listing 3.4, is called. Labels for instructions that have not yet been generated will not be able to be resolved, and so any instruction that references unknown labels will be marked as *invalid* and added to a *pending instructions* vector. In the second pass, all of the *pending instructions* are visited, trying to resolve all remaining labels. At this point, all labels should be able to resolve, since all of the `OPBFOp` objects have been created.

Once all the instructions have been created, all of the information is written into a binary file. It should be noted that neither the binary OPBF file nor the format itself is optimized for hardware execution, as is probably the case with the NVIDIA Cubin format. It is solely intended for emulation purposes. There are several parts to the OPBF file: the OPBF Header, which includes the OPBF Kernel Headers, the code, the global descriptors, variable initialization data, the kernel descriptors, and the label table.

The general OPBF file header is shown in Listing 3.5, and is the first information found in the file. It contains a vector of kernel structures, as shown in Listing 3.6, which contain basic information for each kernel in the file, such as the kernel name or the entry point. The descriptors (both global and kernel descriptors) contained in the OPBF file are structured as shown in Listing 3.7. These hold information for the size of each descriptor, which correspond to the memory spaces defined in the PTX specification.

The state spaces defined by PTX are registers (.reg), special registers (.sreg), constant memory (.const), global memory (.global), local memory (.local), parameters (.param), shared memory (.shared), surface memory

(.surf), and texture memory (.tex). The OPBF file adds an extra space for the kernel code (.code). The different memory spaces correspond to all the available memory spaces available in the GPU architecture, as described in Section 2.2. The parameters space is used to access user parameters.

```
struct opbf_header {
   uint32_t nkernels; /* Number of kernels in the file */
   struct opbf_kernel kernels[nkernels];
};
```

Listing 3.5: *OPBF Header Structure*

```
struct opbf_kernel {
   char name[128]; /* Kernel name */
   uint32_t entry; /* Entry point for the kernel */
   uint32_t nparams; /* Size of the kernel parameters */
   uint32_t offset; /* Offset inside the file for the init data */
   uint32_t params[nparams]; /* Size of each parameter */
};
```

Listing 3.6: *OPBF Kernel Structure*

```
struct opbf_descriptor {
   uint32_t address;
   uint32_t size;
};
```

Listing 3.7: *OPBF Descriptor Structure*

```
struct op_common {
   uint8_t format:4; // Instruction format
   uint8_t sreg:2; // Number of source register
   uint8_t dreg:2; // Number of destination registers
   uint8_t pred:1; // Has predicate?
   uint8_t npred:1; // Negated Predicate?
   uint8_t imm:1; // Has immediate value?
   uint8_t op:5; // Operation number
};
```

Listing 3.8: *OPBF Opcode Common Structure*

The OPBF code section contains all of the kernel code, the parsed PTX instructions. All OPBF opcodes contain two sub-structures: a common structure for all opcodes (shown in Listing 3.8 and in Figure 3.10) and a

specific structure for each opcode class. The common part of the structure stores basic information common to any opcode: the instruction format (`OPBFRs`, `OPBFRbs`, `OPBFCb`, `OPBFMem`, `OPBFRst`, `OPBFGet`, `OPBFUni` or `OPBFOb`), the number of source and destination registers, if the opcode has a predicate, if that predicate is negated, if there is an immediate value and the exact opcode number.

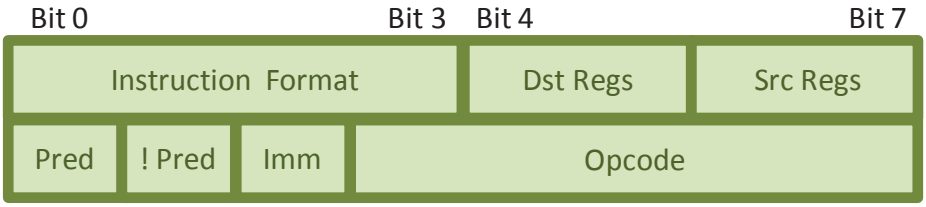

Figure 3.10: *OPBF Opcode Common Header*

After the common opcode header comes a format-specific opcode structure. Listing 3.9 shows the specific header for an Rbs opcode, which is the most common one. The rounding, bit and saturation modifiers are included in the specific opcode header and not in the common header because not all opcode formats have these attributes. By doing this, the code size is reduced significantly. The operand type contains a 4-bit encoding of the source and destination operand types, following the *tiss* format: type (int, float), sign (signed, unsigned) and size (8, 16, 32, 64). Thus, for example, 0111 is a signed 64 bit integer. After the specific opcode header, the operands are written, such as the source and destination registers, the predicate and the immediate values.

```
struct op_rbs {
   uint8_t rnd:2; // Rounding Modifier
   uint8_t bit:2; // Bit Modifier
   uint8_t sat:1; // Saturation Modifier
   uint8_t type:4; // Operand Type
   uint8_t rev:7; // Reserved
};
```

Listing 3.9: *`OPBFRbs` Opcode Specific Structure*

The OPBF files are particularly small due to the highly optimized data structure used. Therefore, the matrix multiplication kernel, for example, occupies 1856 bytes, while the cubin file for the same kernel occupies 2861 bytes. All of the tested kernels, as described in Section 4.1, are at least 30% smaller than their Cubin counterparts.

```
0x00000000 [9] _cvt._rn._s32._u16 r4108 r20
0x00000009 [13] _mul24._rn._hi._s32 r4188 r4108 (4) 16
0x00000016 [9] _cvt._rn._s32._u16 r4128 r22
0x0000001f [11] _ld.__cons._s32 r4132 (4) 0xc002000c
0x0000002a [11] _mul._rn._lo._s32 r4096 r4128 r4132
0x00000035 [13] _mul._rn._lo._s32 r4184 r4096 (4) 16
0x00000042 [11] _add._rn._hi._s32 r4096 r4184 r4132
0x0000004d [13] _sub._rn._hi._s32 r4172 r4096 (4) 1
0x0000005a [9] _cvt._rn._s32._u16 r4156 r8
0x00000063 [9] _cvt._rn._s32._u16 r4120 r10
0x0000006c [11] _ld.__cons._s32 r4112 (4) 0xc0020010
0x00000077 [11] _setp._lt._s32 r4100 r4172 r4184
0x00000082 [11] _mov._rn._hi._f32 r4096 (4) 0f00000000
0x0000008d [9] @r4100 _bra (4) 0xe00004d5
```

Listing 3.10: *matMul OPBF code fragment*

A disassembly tool was developed for debugging purposes, *opbfdis*, taking an OPBF file as input and displaying the opcodes, operands, modifiers, and the global kernel descriptors. This was very straight forward to do, thanks to the object oriented approach taken. Listing 3.10 shows the first 14 lines of output of the matrix multiplication kernel. These correspond to the PTX instructions as shown in Listing 3.1. Note the absence of labels, such as in the `ld`, `mov` and `bra` statements.

3.5 Emulation Library (libemul)

Once an OPBF file has been created from a PTX source, it can be loaded by the emulation library *libemul* and executed, obtaining the same output as executing on the GPU. *libemul* has several classes that model each architectural component, as described in 2.2.

The `Card` class models a GPU, and contains objects and properties that model different parts of the architecture, such as one or more multiprocessors (`MP` class) or the global, texture and constant memory spaces. It provides methods such as allocating global memory, translating device addresses to host addresses or loading OPBF files, to the instantiating entity, which will be described in Section 3.6. The `MP` class models a streaming multiprocessor, and so it contains a number of processors, each with its own register and local memory. The *shared memory* space is also emulated at this level. Each `MP` object contains a list of `Core` objects. The `Core` class

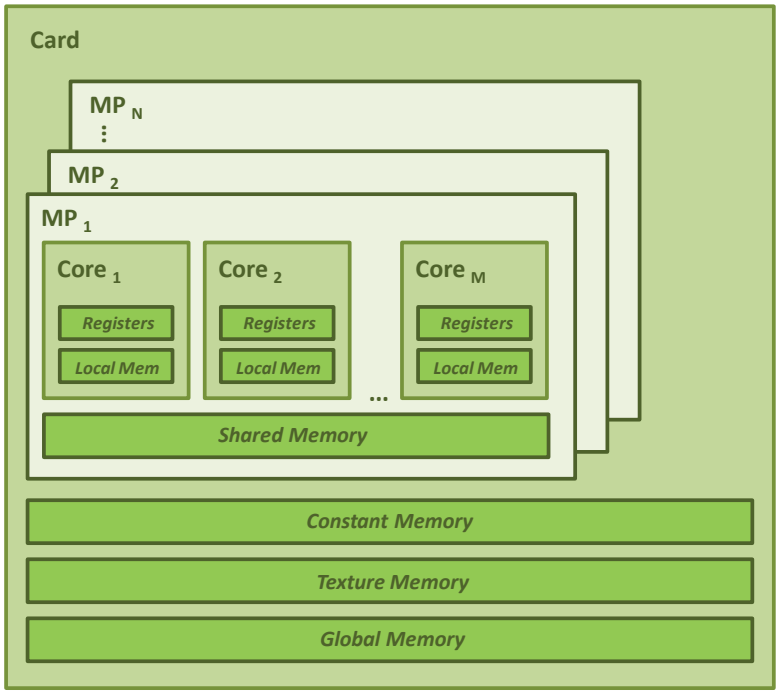

Figure 3.11: *libemul General Diagram*

models a scalar processor, as defined by the architecture. It contains a set of registers and local memory, program counter, and thus contains the code belonging to opcode execution. This hierarchy is shown in a graphic and clearer way in Figure 3.11. Note how this is entirely based on the CUDA thread hierarchy, as seen in Figure 2.2.

The emulation library is multi-threaded, using the standard POSIX threads library *pthreads*. If a multi-core CPU is available for execution, the `ECUDA_THREADS` environment variable can be used to set the number of `MP` objects that will be instantiated. Thanks to the CUDA architecture, the separation into different threads is straight forward: the total number of blocks can be divided among all the available multiprocessors. Thus, in our implementation, if T threads are available, `MP` object N works with thread blocks $T*k+N$. If 3 cores were available and therefore 3 `MP` objects were instantiated, the first `MP` object (0) would execute blocks (0, 3, 6, ...), the second `MP` object (1), would execute blocks (1, 4, 7, ...) and the third `MP` object (2), would execute blocks (2, 5, 8, ...). In an `MP` instance, the blocks are executed sequentially. This block separation into threads allows for a very efficient parallelization, as will be discussed in Section 4.4.

All of the memory spaces defined in the CUDA architecture are imple-

Start	End	Section	Description
0x00000000	0x00000fff	.sreg	Special Registers
0x00001000	0x00002fff	.reg	General Purpose Registers
0x00003000	0xbfffffff	.global	Global Memory
0xc0000000	0xc000ffff	.const	Constant Memory
0xc0010000	0xc001ffff	.local	Local Memory
0xc0020000	0xc002ffff	.param	Parameters/Stack
0xc0030000	0xc00fffff	.shared	Shared Memory
0xc0100000	0xd007ffff	.surf	Surface Memory
0xd0080000	0xdfffffff	.tex	Texture Memory
0xe0000000	0xffffffff	.code	Code Memory

Table 3.3: *libemul Device Memory Map*

mented in the emulation library. In fact, all of the hardware resources, including registers and special registers are included in the memory map, as shown in Table 3.3. The texture, constant and global memory are stored in the `Card` class, while the `MP` class holds the shared memory and the `Core` class holds the registers and local memory, including the special registers. All of these spaces are simply implemented as a data vector, except for the special registers, which are stored as properties of the `Core` class. It should be noted that, although all data type size (8, 16, 32 and 64 bits) registers are supported, internally they are all stored as 32-bit or 64-bit registers, in order to simplify the allocation process and ease address translation.

Each of the classes in the *emul* hierarchy implement a translation method (`translate()`) that translates *device* addresses to *host* addresses. The address translation is straight forward: *host memory space vector start address* + (*device address* - *device space start address*). Each class resolves the addresses for the spaces for which it stores the actual data. Thus, for example, the `Core` class handles translations for normal and special registers and local memory. Other cases are handled by calling the `MP` class method, which behaves in the same way. Thus, for example, the `_ld.__cons._s32 r4132 (4) 0xc002000c` instruction (taken from Listing 3.10, line 4) requires two device address translations: one for register 4132 and another for constant space address 0xc002000c. The first one will be translated directly by the `Core` class translation, while the constant memory address will be translated by the `MP` class.

After taking into consideration all of the emulation techniques described in Section 2.4, a refined interpretation method was chosen. The main reason

for this was the excessive complexity of implementing *binary translation* for x86, which would be a non-trivial task because of the important differences between the x86 ISA and the PTX ISA and the two architectures.

In order to reduce the amount of lookups during execution, a *code cache* was implemented. When loading a kernel, an encapsulating object of the `Opcode` class is created for each OPBF instruction. A lookup function, `getExe()`, performs a lookup on a vector by the opcode number, available via the `OPBFOp` object, returning a pointer to a method of the `Core` class that actually implements the given opcode. The `Opcode` object contains the `OPBFOp` object and the previously mentioned pointer. The *code cache* is implemented as an STL map container, storing the program counter value for an opcode as the key and the `Opcode` object as the value. Therefore, during execution, no interpretation takes place. When a thread reaches a certain execution point, the *code cache* provides a fast lookup of which function should be executed by looking up the program counter value. The *code cache* is analogous to the *trace cache* found in the Intel Pentium 4 architecture, where instructions that have already been fetched and decoded are stored in a cached, although trace caches store one or several BBLs, not individual instructions [39].

The main execution loop for a kernel is implemented in the `MP` class. In the `execute()` method, we iterate over the `Core` object array, calling the `Core::execute()` method. Originally, this method performed a lookup in the *code cache* via its program counter to get `Opcode` object and then execute the opcode method. This was improved in later versions. Taking into account the fact that all cores run in parallel, and therefore, they are usually all at the same execution point, the `MP::execute()` method can perform a single *code cache* lookup and pass the resulting `Opcode` object to all cores.

An opcode can access different operand types (register, immediate, memory, address) and different data types (floats, integers). As mentioned above, the actual methods that implement all of the PTX ISA opcodes are available in the `Core` class. In order to handle all operand and data types, a lot of conditional code was originally added to the execution functions. This proved to be extremely inefficient, as the operand and data types for an opcode are static. Therefore, having conditional code to distinguish the cases at runtime is inefficient and redundant, as the result will not vary.

C++ *templates* were introduced to reduce the amount of code needed to implement an opcode and to eliminate the need to run type distinguishing code at runtime. [40]. It is a feature of C++ that allow functions to operate

with generic types. That is, to code a single function instead of having to code different functions for different data types or to perform runtime type-decision code. Listing 3.11 shows a basic example of a *template*. With a single function, any two operands can be compared. The calling syntax includes the type, so a call to `max<float>(4.5, 8.5)` would invoke `max` function with `float` operands. This is only an example, as an operation as simple as calculating the maximum of two values could be achieved with a much simpler preprocessor macro.

```
template <typename T>
T max(T a, T b)
{
  if(a > b)
    return a;
  else
    return b;
}
```

Listing 3.11: *C++ template example*

When a C++ program with *templates* is compiled, on every call to a templatized function, the compiler creates a copy of the function with the templatized parameters. On the previous example, the compiler would generate a copy of the `max` function with `double` parameters. Therefore, the use of *templates* increases the binary size, but does not add any runtime overhead.

In our case, an opcode template will have a template parameter for every operand type, and one or several template parameters based on the opcode data types. For example, the `mul` opcode (Listing 3.10, line 5) has a data type parameter and three operand type parameters. The data parameter can be an 8, 16, 32 or 64 bit signed or unsigned integer, a float, or a double. The first and second operand types are always registers (the destination and source registers, respectively), but the third operand (the second source operand for the multiplication) can either be a register or an immediate. All of these combinations must be supported, since an opcode could potentially use any of them, and that information will only be available when a kernel is loaded.

A fragment of the code that decides, at runtime, which `mul` opcode implementation should be invoked is shown in Listing 3.12. This code is very repetitive and it can be automatically generated from a data type list and small amounts of information available for each opcode. This is

accomplished by a *Perl script* that is executed on compilation time, which generates a function similar to the one shown in Listing 3.12 for each opcode. It should be noted that these functions take an *OPBFOp* operand as a parameter, which is used for obtaining the operation type (`op.type()`), or if the opcode has an immediate value (`op.hasImmediate()`).

```
Exe Core::ExeMul(const OPBFOp &op) {
   switch(op.type()) {
      case _u32:
         if(op.hasImmediate())
            return Mul<uint32_t, reg, reg, imm>;
         else
            return Mul<uint32_t, reg, reg, reg>;
      case _s32:
         if(op.hasImmediate())
            return Mul<int32_t, reg, reg, imm>;
         else
            return Mul<int32_t, reg, reg, reg>;
      case _s16:
         if(op.hasImmediate())
            return Mul<int16_t, reg, reg, imm>;
         else
            return Mul<int16_t, reg, reg, reg>;
      case _u16:
         if(op.hasImmediate())
            return Mul<uint16_t, reg, reg, imm>;
         else
            return Mul<uint16_t, reg, reg, reg>;
      case _s8:
         if(op.hasImmediate())
            return Mul<int8_t, reg, reg, imm>;
         else
            return Mul<int8_t, reg, reg, reg>;
...
```

Listing 3.12: *`Mul` Template Invocation*

These functions are only executed when generating the *code cache*, since the type parameters are fixed and cannot change at runtime. This has a strong positive impact on performance, since once the *code cache* is built, a pointer is available to the template version of the opcode execution function for each opcode. Therefore, once a kernel execution launches, no more type

comparisons are necessary.

This approach also significantly reduces the amount of code needed to actually implement an opcode. Listing 3.13 shows the complete code of the function that implements the `mul` opcode. Thanks to templatization, the code is extremely straight forward. After obtaining pointers to the operands, they are simply multiplied and stored back. Then, the value of the *program counter* is incremented. This process makes opcode implementation straight forward. Because of this, about 60% of the opcodes defined in the PTX ISA have been implemented. Implementing the rest of them should be trivial.

```
template<typename T, operand_t D, operand_t A, operand_t B>
static void Mul(Core *core, const OPBFOp &op) {
   T *a, *b, *d;
   getValues<T, D, A, B>(core, op, d, a, b);
   *d = *a * *b;
   core->incPC(op.size());
}
```

Listing 3.13: *`Mul` Executor Implementation*

`getValues()` is a templatized functions that obtains pointers to the opcode operands. If the `mul` operand executor function shown above was called with data type `T` and operands `D`, `A`, and `B`, the `getValues()` function with the same parameters would be invoked. It allows a single function call to obtain operands as different as an immediate, a register or a memory location. Listing 3.14 shows the simplest getValues() implementation, with only two operand parameters. It contains calls to templatized functions `getSource()` and `getDestination()`, which retrieve pointers to the source and destination operands, respectively.

```
template<typename T, operand_t D, operand_t A>
static inline void getValues(Core *c, OPBFOp &op, T *&d, T *&a) {
   a = CoreExtractor<T, A>::getSource(*c, op, 0);
   d = CoreExtractor<T, D>::getDestination(*c, op, 0);
}
```

Listing 3.14: *`getValues()` Implementation*

Since the actual retrieval of the source and destination operands is different if an operand is a register, a memory address or an immediate value, complete *template specialization* is done, providing different versions of the `getSource()` and `getDestination()` functions. The register versions simply call the translation methods to obtain the host address of the register.

The immediate version simply obtains the immediate location, which is stored in the `OPBFOp` object. The indirection version first obtains the address of the register that stores the address, obtains the value, and translates it to a host address. The address version simply returns the translation of the address, which is stored as an immediate and retrieved in the same way.

3.6 CUDA Runtime Library (libecuda)

The *CUDA Runtime Library libecuda* serves as a replacement for the *libcudart* runtime library provided by NVIDIA. It implements several methods of the documented CUDA API, such as `cudaMalloc` or `cudaMemcpyToArray`, which basically perform execution control and memory management. Several undocumented methods such as `__cudaRegisterFunction` or `__cudaRegisterTexture` have also been implemented, as calls to them are inserted into the host code by the *nvcc* compilation flow.

When a CUDA application with the Cubin data included in the host code is started, as explained in the chapter overview, a kernel call is translated into a conventional C function call which acts as a wrapper to several API function calls. Several other API calls are inserted before and after the kernel call. All of these functions are implemented in the NVIDIA CUDA runtime library, and therefore they must all be reimplemented, or the application would not link correctly.

An initialization function `__init` is defined in the library, which is loaded before the program execution starts. In this function, a `Card` object (as described in Section 3.5) is instantiated. The current directory is searched for OPBF kernel files, which are loaded using the `Card::addFile()` method. The `ECUDA_PATH` environment variable can also be used to define which directories should be searched for OPBF files. The `ECUDA_THREADS` environment variable is also read at this point, setting the number of threads that will be used to run the emulation. The initialization function is needed because there will be numerous function calls prior to the kernel execution that will requiere the `Card` object to be loaded. There is an equivalent `__fini` function that calls the `Card` destructor, freeing allocated memory.

```
[__cudaRegisterFatBinary] : 0x8059f80
[__cudaRegisterFatBinary] : 0x8059f40
[__cudaRegisterFunction] : 0x804b35c -> _Z9matrixMulPfS_S_ii
[cudaGetDeviceCount] : 0xffaff498 (1)
[cudaGetDeviceProperties] : Device 0
[cudaSetDevice] : 0
[cudaMalloc] : 393216 bytes, 0x3000 [0xf7908008]
[cudaMalloc] : 524288 bytes, 0x64000 [0xf7887008]
[cudaMemcpy] : 0x3000 -> 0xf79ea008 (393216 bytes) (HostToDevice)
[cudaMemcpy] : 0x64000 -> 0xf7969008 (524288 bytes) (HostToDevice)
[cudaMalloc] : 786432 bytes, 0xe5000 [0xf77c6008]
[cudaConfigureCall] : grid (32,24,1) block (16,16,1)
[cudaSetupArgument] : 4 @ 0
[cudaSetupArgument] : 4 @ 4
[cudaSetupArgument] : 4 @ 8
[cudaSetupArgument] : 4 @ 12
[cudaSetupArgument] : 4 @ 16
[cudaLaunch] : _Z9matrixMulPfS_S_ii
[cudaMemcpy] : 0xf7705008 -> 0xe5000 (786432 bytes) (DeviceToHost)
Processing time: 17092.416016 (ms)
Test PASSED
[cudaFree] : 12288
[cudaFree] : 409600
[cudaFree] : 937984
[__cudaUnRegisterFatBinary] : 0x26
[__cudaUnRegisterFatBinary] : 0x26
```

Listing 3.15: *matMul Runtime Library calls*

Listing 3.15 shows the complete output when running the matrix multiplication benchmark in `DEBUG` mode, which shows all calls to runtime library functions. The usual flow of a CUDA application is:

1. Calls to device and kernel registration and initialization functions, such as `__cudaRegisterFatBinary()`, `cudaGetDeviceCount()`, `cudaGetDeviceProperties()`, and `cudaSetDevice()`. Most of these calls are not directly made by the CUDA developer. They are inserted by the *nvcc* compiler.

2. Space allocation on the device, such as the `cudaMalloc()` call in the above example.

3. Data copying, where the data which the kernel must process is copied from the host to the device, such as the `cudaMemcpy()` call in the above example.

4. Kernel preparation and launching, done by the `cudaConfigureCall()`, `cudaSetupArgument()` and `cudaLaunch()` calls. The kernel is actually launched after the call to the `cudaLaunch`, which triggers a call to the `Card::SetupCall()` method, which as explained in Section 3.5, actually launches the kernel in emulated mode.

5. After the kernel execution finalizes, the calculated data must be copied back to the host, as shown by the `cudaMemcpy()` call in the above example.

6. Allocated data is freed, and destructors are called, as shown by calls to the `cudaFree` and `__cudaUnRegisterFatBinary` functions.

The implemented runtime library functions can be categorized as follows:

- **Initialization functions**

  `__cudaRegisterTexture`, `__cudaRegisterFatBinary`, `__cudaUnregisterFatBinary`, `__cudaRegisterFunction`, `__cudaRegisterShared`, `__cudaRegisterVar`

  These functions are undocumented by NVIDIA, and their intended usage can only be deduced by the function name and parameters. The use of some of them is unknown, and they are implement as a blank function, to allow CUDA applications to link correctly. Most of them are simply implemented as storing the passed values in a data structure to allow later retrieval. For example, the `__cudaRegisterTexture` maps a texture name to a pointer to an unknown data structure. This pointer value must be saved, as future references to texture in other API functions are not done by texture name but by this pointer value.

- **Memory Allocation Functions**

  `cudaMalloc`, `cudaFree`, `cudaMallocArray`, `cudaFreeArray`

  These functions are documented in the NVIDIA CUDA API [15]. They are basically wrappers around the functions `Card::Malloc()` and `Card::Free()`.

- **Memory Copy Functions**

  `cudaMemcpy2DToArray`, `cudaMemcpyToArray`, `cudaMemset`, `cudaMemcpy`, `cudaMemcpyToSymbol`

  These functions basically operate by obtaining the device address, by

calling `Card::translate()`, and performing regular recalls to memory copy operations.

- **Texture Functions**

  `cudaBindTextureToArray`, `cudaUnbindTexture`, `cudaGetChannelDesc`

  These functions allow texture memory access, by associating the texture device address to the global memory address where the data is stored.

- **Kernel Launch Functions**

  `cudaConfigureCall`, `cudaSetupArgument`, `cudaLaunch`

  The first two of these functions copy the kernel parameters to a known area, so the device emulation will be able to obtain them and make them available as the *param* memory space.

- **Device Properties Functions**

  `cudaGetLastError`, `cudaGetDeviceCount`, `cudaSetDevice`, `cudaGetDeviceProperties`

  These functions hide the fact that no GPU card is actually running the kernel by returning, for example, the physical parameters (such as global memory size or total number of registers) about the emulated GPU card.

Chapter 4

Experimental Evaluation

Once an initial working version of the emulation library had been developed, in order to evaluate the correctness of the implemented design, several unmodified CUDA benchmarks were run through the emulation process. Several major bugs were found, as mentioned in Chapter 3, some of which crashed the emulator, and some of which simply prevented the output from being the same as in a native GPU. All of them were fixed, making the emulated output equivalent to the GPU output. Performance measurements were then taken, comparing the *libecuda* emulation times against equivalent CPU-only algorithms, the GPU, and *device emulation* mode. Finally, profiling measurements were taken to identify major bottlenecks in the current implementation.

4.1 Benchmark Descriptions

The chosen benchmarks were *matMul*, *cp*, *tpacf*, *rpes*, and *sad*. Table 4.1 shows a list of the benchmarks, with their grid dimensions and total number of threads. The *matMul* benchmark was chosen as the first benchmark due to its simplicity. The four remaining benchmarks were taken from the Parboil Benchmark Suite [41, 8], developed by the IMPACT Research Group at the University of Illinois at Urbana-Champaign. It offers several benchmark that are better suited to measure a GPU platform performance than traditional benchmark suites. They are all different in the memory spaces they use, the PTX instructions that are generated from the source code, and the runtime calls they make.

- The Matrix Multiplication (*matMul*) benchmark implements matrix multiplication. It has been extracted from the NVIDIA CUDA SDK,

Benchmark	Grid Dim	Block Dim	Total Threads
matMul	(32,24,1)	(16,16,1)	196608
cp	(4,61,1)	(16,8)	32768
rpes	(65535,1,1)	(64,1,1)	4194240
sad	(44,36,1)	(61,1,1)	96624
	(44,36,1)	(61,1,1)	96624
	(11,9,1)	(32,4,1)	12672
	(11,9,1)	(32,1,1)	3168
tpacf	(201,1,1)	(256,1,1)	51456

Table 4.1: *Benchmark Properties*

and is one of the most basic CUDA examples available. It uses *shared* and *param* memory. The generated PTX uses the `add` (addition), `bra` (branching), `cvt` (data type conversion), `ld` (load data from memory to a register), `mad` (multiply and add), `mov` (set a register value from an immediate value or another register), `mul24` (24-bit multiplication), `mul` (multiplication), `setp` (compares two values with a relation operation, optionally combining the result with a predicate value, writing the result to a register), `st` (copy data from a register to memory) and `sub` (subtract) instructions. In our tests, the two multiplied matrices have a size of 384x256 and 256x512, yielding a 384x512 matrix as a result. The source matrices are generated randomly at the start of the program.

- The Coulomb Potencial (*cp*) benchmark measures the electric potential at each point in a 3D volume in which charges are randomly distributed. In our tests, the atom count is 40000 and the volume size is 512x512, the default values. Besides using the *param* memory, it introduces the use of the *const* memory space. The only additional instruction used is `rsqrt`, which performs the inverse of a square root.

  The benchmark uses the `cudaMemcpyToSymbol()` API call, which allows copying data from a host variable to a device variable. Since it references the device variable by its name, it order to properly support the call, we had to implement keeping a label list in the OPBF file, allowing the runtime library access to it, to obtain the device address of the symbol.

- The Two-point Angular Correlation Function (*TPACF*) describes the

angular distribution of a set of points. It is used to calculate the probability of finding two points separated by a given angular distance. It computes the angular correlation function for a data set of astronomical bodies. The benchmark was used with its original data input and all of its default parameters. It uses the *param*, *const* and *shared* memory spaces. Support was added for several instructions which it introduced, such as `div` (division), `neg` (arithmetic negation), `or` (bitwise or), `selp` (selects between source operands, based on the predicate source operand), `set` (very similar to the `setp` instruction), `shl` (binary shift-left), `shr` (binary shift-right).

The `selp` instruction is the only PTX instruction that allows immediate values, although this is not clearly stated in the PTX specification. This caused problems in our initial implementation, which assumed that there was only one immediate value per instruction. Also, this benchmark builds a complex CFG, as shown in Section 3.3, which caused register allocation problems, leading to a complete rewrite of the register allocation algorithm. Selp has two immediates.

- The Rys Polynomial Equation Solver (*rpes*) calculates 2-electron repulsion integrals, a molecular dynamics sub-problem, which represent the Colomb interaction between electrons in molecules. The benchmark was used with its default values, 2 atoms and 5 shells. This was the first benchmark that introduced the use of the `texture` memory space, besides *shared* and *param* memory. Support was added for the new instructions `and` (bitwise and), `tex` (texture access), `ex2` (exponentiate a value in base 2), `rcp` (take the inverse of a value) and `abs` (take the absolute value). The `ld` instruction was used in vectorized form, writing to four registers in a single instructions, and support for this had to be added to the PTX parsing and the OPBF code generation. Support for 1D textures was added to support this benchmark. Finally, it required barrier support to be properly implemented. The *matMul* benchmark also used barrier calls, but they were unnecessary.

- The computation of Sums of Absolute Differences (*sad*) benchmark is based on the full-pixel motion estimation algorithm found in the reference H.264 video encoder. It basically searches for blocks in one image that approximately match blocks in another image. It computes sums of absolute differences for pairs of blocks, representing how similar they are. The benchmark is composed of three kernels:

host	CPU	GPU
hercules	Intel Core 2 Duo 6600 at 2.40GHz, 4GB RAM	GeForce 8800 GTX
obelix	2 x Dual Core AMD Opteron 2222 at 3.0GHz, 8GB RAM	GeForce GTX 280
praline	2 x Intel Quad-Core Xeon E5420 at 2.50GHz, 8GB RAM	Tesla S870

Table 4.2: *Benchmark Testing Hardware*

one of them computes SADs for 4x4 blocks, another one takes the results to compute SADs for up to 8x8 larger blocks, while the last one computes SADs for blocks up to 16x16. The two source images used are the default ones, which are two QCIF-size images (176x144) over a 32 pixel block search area.

The benchmark uses uses the *shared*, *param*, and *tex*. Support was added for the `rem` instruction, which calculates the remainder of an integer division. The benchmark used 2D textures, which required expanding the previous texture support, and it took advantage of *texture address clamping*, where coordinates wrap around the maximum and minimum values, which also required explicit code changes. It also uses *dynamic shared memory*, which required changes to the PTX parsing, as it generated zero-sized shared memory vectors, and support for the `__cudaRegisterShared()` runtime call, which makes the environment aware of the *dynamic shared memory* usage. The `st` instruction was used in vectorized form, requiring similar changes to those supporting `ld` vectorized instructions.

4.2 Testing Conditions

Performance measurements for the benchmarks were done on three different machines, whose characteristics are summarized in Table 4.2. These machines were selected due to the disparity of their configuration. Two of them, *hercules* and *praline* are Intel-based, while *obelix* is AMD-based. *hercules* is capable of running 2 threads in parallel, *obelix* is capable of running 4, and *praline* is capable of running 8.

The three systems are equipped with GPUs, with *obelix* being the most powerful. The GPUs in *hercules* and *praline* are essentially the same, although the Tesla S870 found in *praline* actually contains two C870 cards in an external rack. The rack is connected to the server through a PCI Express extension cable. The C870 card is essentially a GeForce 8800 card without video output. Even though *praline* and *obelix* contain two GPUs

each, only one of them will be used, since the tested benchmarks do not support multiple GPU cards.

Four versions of each of the benchmarks were compiled, all of them with the maximum available compiler optimizations (`gcc -O3`).

- A CPU version, with the original non-parallelized, non-GPU version of the benchmark. It should run slower than the GPU versions, but faster than the emulated GPU versions. It will be tested on the three available hosts.

- A GPU version, running on the three available GPUs. It should yield, by far, the smallest execution time, and will serve as the basis to measure the other versions slowdown. The fastest execution should occur on *obelix*.

- A *deviceemu* emulated GPU version, running the previous version in an emulated environment, without using the GPU. It is expected to be hundreds or thousands of times slower than the native GPU version. It will be tested in the three available hosts. The fastest execution should be on *praline* or *obelix*, which can achieve better performance than *hercules*.

- A *libecuda* emulated GPU version, running the GPU version in our own emulated environment, without using the GPU. It is also expected to be hundreds or thousands of times slower than the native GPU version. Since we can control the number of threads launched, we can evaluate how well *libecuda* scales by running the test in *hercules* with 1 and 2 threads, in *obelix* with 1, 2 and 4 threads, and in *praline* with 1, 2, 4 and 8 threads. The fastest result should be delivered by *praline* with 8 threads.

The three testing systems run Debian GNU/Linux testing, and all are running the 2.6.26-2-amd64 kernel version. In order to obtain the most accurate possible timing measurements, when running the *deviceemu*, *libecuda*, and CPU versions of the benchmarks, the systems were running in *single-user mode*. In this mode, no network interfaces are available, no daemons are running, and no other users besides *root* are allowed login. Running the GPU versions in *single-user mode* is not straight forward, since there are numerous problems with the NVIDIA drivers when running in that mode. Therefore, this test was not performed in *single-user mode*, but the test was repeated numerous times, and the results did not change significantly.

Benchmark	Exact?	Error
matMul	No	$< 10^{-6}$
cp	No	$< \max(0.5\%, 0.005)$
rpes	No	$< 10^{-9} + 2\%$
sad	Yes	
tpacf	No	$< 10\%$ (first 3 bins), $< 1\%$ (rest)

Table 4.3: *Benchmark Functional Results*

The timing of the *deviceemu*, *libecuda*, and GPU versions was done by using the benchmark provided timers, which use a timing subsystem from the Parboil suite. The matrix multiplication benchmark uses a timing mechanism from the CUDA *cutil* library. The parboil subsystem timers give the results broken down into IO, GPU, Copy and Compute times. These times were all taken into consideration, since in fast benchmarks, the IO and Copy times are much larger than the GPU processing time.

4.3 Functional Results

All of the benchmarks described in Section 4.1 can be run within the *libecuda* emulation with their default data set, returning the expected output values. These, however, are not always *exactly* the same, when comparing CPU, GPU and emulated output. This is mainly due to different IEEE floating point implementations between the CPU and the GPU, as noted by NVIDIA [15]. Emulating this behavior would mean a performance hit, and it would require very precise details on the floating point implementation, which are not available. Also, several rounding modifiers described in the PTX specification are currently ignored in our emulation process, which could lead to accumulated rounding errors.

Table 4.3 shows the functional results of the five tested benchmarks. Only the *sad* benchmark produces exact results, since it does not use floating point operations. The rest of the benchmark make heavy use of floating point operations, which yield different results when executed on a CPU than on a GPU. This has been acknowledged by NVIDIA [15], since not even their own *device emulation* produces the same results as the GPU.

The error values in Table 4.3 are obtained from the `compare-output` Python script, which is included in all of the benchmarks in the *Parboil* suite. The script compares the benchmark output with a *reference* output

CPU	GPU	deviceemu	libecuda
9.092870e-03	9.092866e-03	9.092870e-03	9.092866e-03
1.973599e-02	1.973597e-02	1.973599e-02	1.973597e-02
2.776850e-02	2.776849e-02	2.776850e-02	2.776849e-02
1.569516e-02	1.569517e-02	1.569516e-02	1.569517e-02
2.591271e-02	2.591271e-02	2.591271e-02	2.591271e-02
1.546891e-02	1.546890e-02	1.546891e-02	1.546890e-02
2.582114e-02	2.582114e-02	2.582114e-02	2.582114e-02
5.472301e-13	0.000000e-00	5.472301e-13	0.000000e-00
3.879561e-11	0.000000e-00	3.879561e-11	0.000000e-00
2.795462e-06	2.793545e-06	2.795462e-06	2.793545e-06

Table 4.4: *rpes benchmark Results Comparison*

calculated using the CPU-only versions of the benchmarks, and checks if the produced values are within a tolerable margin. For the *matMul* benchmark, taken from the CUDA SDK, the error value was obtained from the benchmark source code, where the obtained result is also compared against a CPU-only obtained solution.

The error reaches large values, such as 10% in the *tpacf* benchmark, due to the fact that the GPU parallelized algorithms are not always exactly the same as the CPU-only algorithms. Different approximations are performed, or in the case of the *tpacf* benchmark, a smaller number of samples is taken for the first part of the calculations. Therefore, there are two main sources of error: CPU vs GPU algorithm differences and floating point calculation differences.

Table 4.4 shows the initial part of the output of the rpes benchmark. NVIDIA *deviceemu* produces exactly the same output as the CPU version of the algorithm, while *libecuda* produces exactly the same output as the GPU version. This is not the same behavior shown on other benchmarks, where none of the four outputs are exactly the same.

4.4 Performance Evaluation

The results of the benchmarks are shown in Table 4.5 and Figures 4.4, 4.5, and 4.6, which show the slowdown of each benchmark version compared to the GPU execution time. There are several conclusions which can be taken from the data.

As expected, the GPU performs spectacularly well, achieving feats such as a 454X speedup on the *cp* benchmark on *hercules*, in comparison to the original CPU version of the benchmark. The values, however, fluctuate too much among machines, and while *obelix* should achieve the greatest speedups, it only does so in the *matMul* benchmark. This can be entirely attributed to the fact that the GPU versions of the benchmarks execute on a practically *negligible* time of only a few microseconds, and therefore, the time spend copying the data to and from the GPU card before and after the calculations is actually longer than the actual processing time. If the benchmark datasets were larger, with several seconds of GPU execution time, the speedup values would be higher. This is also the reason why the *sad* benchmark results show the GPU actually *taking longer* than the CPU.

The *deviceemu* and *libecuda* emulations are very slow, reaching astonishing slowdown values of 7569X and 38746X, respectively. This can be obviously ascribed to two facts: we are trying to emulate an massively parallel architecture (GPU) on a limited parallel architecture (CPU). On top of that, the chosen emulation technique also contributes to the slowness.

The slowdown varies considerably from benchmark to benchmark, with the *sad* benchmark achieving a slowdown of only 2X on *deviceemu* and 4X on *libecuda* on *praline*. These fluctuations are due to the instructions differences between the benchmarks. The *sad* benchmark, for example, does not perform floating point operations, while some of the benchmarks are entirely based on double precision floating point operations. If we compare *deviceemu* and *libecuda*, as seen on Figure 4.3, we can see that *libecuda*, by taking advantage of multiple threads, is much faster than *deviceemu* in several benchmarks, while slower in others.

As it can be seen in Table 4.5 and Figure 4.6, *libecuda* benefits strongly from execution with multiple threads. This can be further seen in Figure 4.1, which shows the speedup as a function of the number of threads in *praline*. For all benchmarks except *rpes*, the speedup is almost linear, with *tpacf* approaching the ideal case. The *rpes* benchmark is clearly below the ideal case, achieving a speedup of about 6.3X with 8 threads. This is probably due to an asymmetrical task distribution. As noted in Section 3.5, thread blocks are distributed between the available threads at startup, each taking a part of the workload. If a thread finishes its part before the others, there would be a processor core sitting idle, while it could be working on the remaining thread blocks.

The *deviceemu* emulation does not take advantage of multiple cores in a machine, as it can be seen by observing that some benchmarks actually run

hercules	**ecuda/1t**	**ecuda/2t**			**deviceemu**	**CPU**
matMul	8006	4047			7096	53
sad	31.0	16.1			3.3	-1.1
cp	3362	1697			452	454
rpes	6512	3678			4671	91
tpacf	38746	19301			2324	29
obelix	**ecuda/1t**	**ecuda/2t**	**ecuda/4t**		**deviceemu**	**CPU**
matMul	10694	5490	2655		17440	101
sad	12.7	6.4	3.4		1.2	-2.5
cp	1301	677	353		90	83
rpes	3595	2417	1193		5883	28
tpacf	30141	14781	7569		4328	20
praline	**ecuda/1t**	**ecuda/2t**	**ecuda/4t**	**ecuda/8t**	**deviceemu**	**CPU**
matMul	8696	4356	2213	1172	7200	48
sad	25.6	13.4	7.1	4.0	2.4	-1.1
cp	1393	712	378	188	86	85
rpes	3521	2073	1029	566	5124	40
tpacf	26767	13426	6763	3194	1697	21

Table 4.5: *Execution Results (Slowdown vs GPU)*

slower in *praline*, where 8 cores are available. This can also be checked by running the `top` tool while the emulation is underway: the CPU occupancy never climbs higher than 150% (with 100% for each CPU core available, *praline* could reach 800%). Also, as it can be seen in Figure 4.2, the execution time when benchmarks are emulated in *deviceemu* does not follow any pattern: *matMul* is fastest in *hercules*, *sad* is fastest in *obelix*, *cp* is fastest in *praline*, *rpes* is fastest in *hercules*, and *tpacf* is fastest in *praline*. Moreover, the execution times in *obelix* are much larger than its counterparts in 3 of the 5 benchmarks.

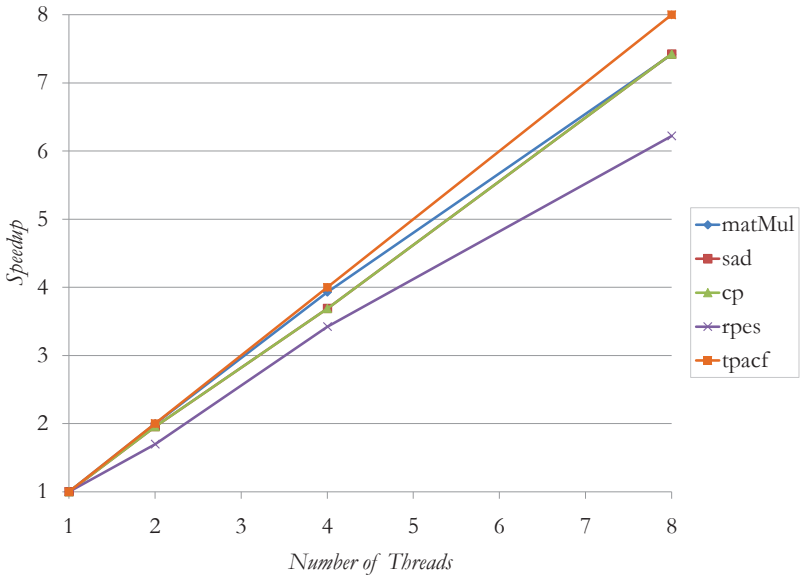

Figure 4.1: *libecuda Scalability Evaluation*

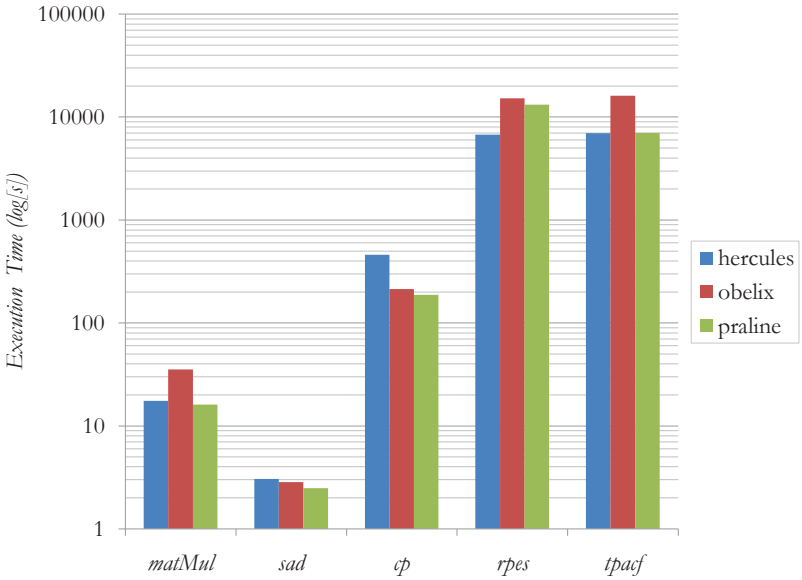

Figure 4.2: *deviceemu Execution Times*

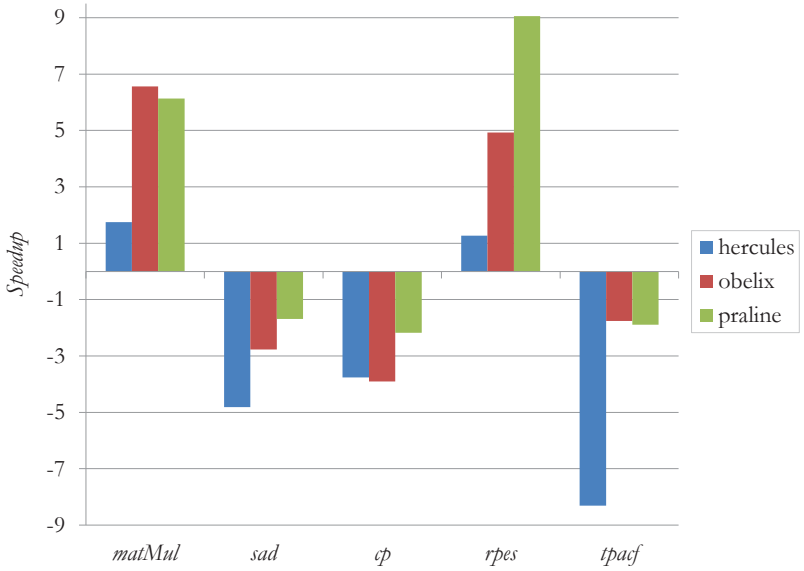

Figure 4.3: *deviceemu vs libecuda Compared Performance*

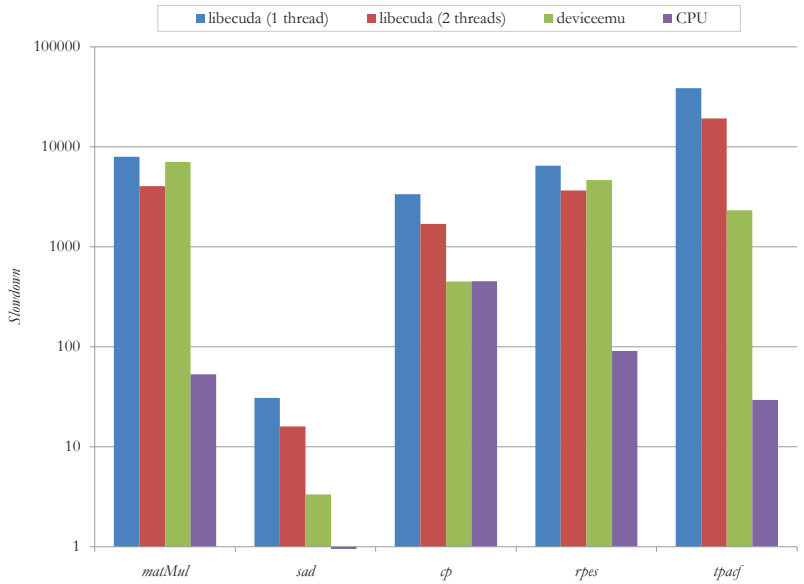

Figure 4.4: *libecuda Performance Evaluation in hercules*

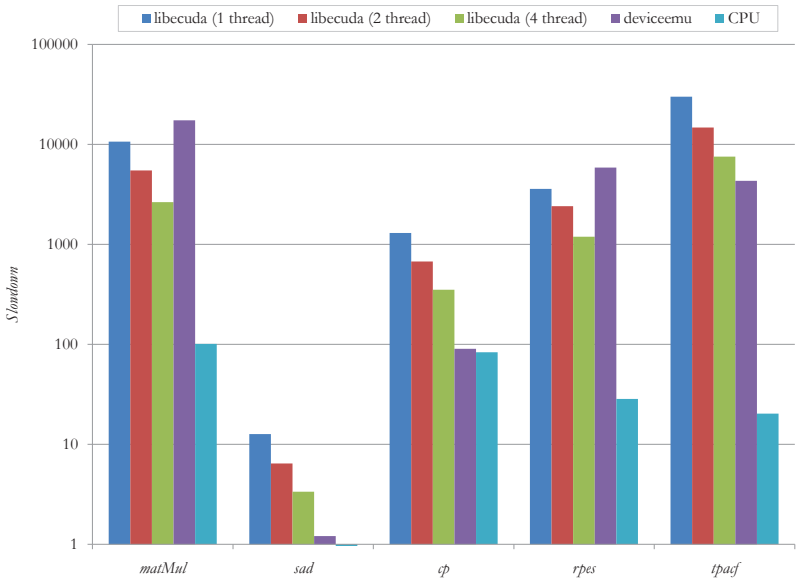

Figure 4.5: *libecuda Performance Evaluation in obelix*

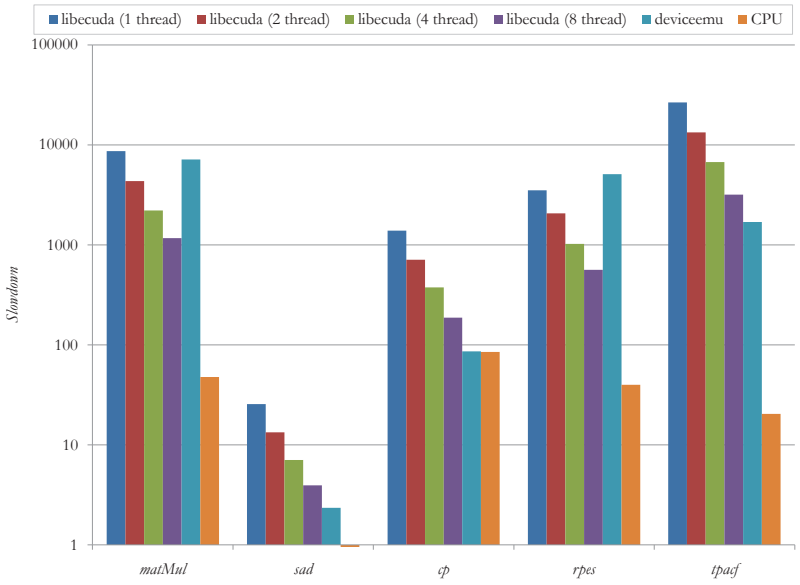

Figure 4.6: *libecuda Performance Evaluation in praline*

4.5 Emulation Library Profiling

Since we wanted to produce exact measurements, the *valgrind instrumenting profiler* was chosen. By using its *callgrind* tool, we can generate a call graph of the functions contained in the application. The five benchmarks were ran through *callgrind* to identify possible bottlenecks and identify code sections which would benefit from a more efficient implementation. Since *callgrind* slows down execution by a factor of at least 400, only the *matMul* and *sad* benchmarks were fully run. The remaining benchmarks were stopped after several hours of execution. This should not alter the results significantly, since the kernel code is basically cyclical.

Incl.	Self	Called	Function	Location
88.29	20.83	2 365 440	Core::execute(FetchEntry c...	matMul: Core.cc, stl_tree.h, ...
18.13	13.93	6 901 760	Core::translate(unsigned lo...	matMul: Core.cc
9.97	9.57	2 324 596	OPBFOp::operandsSize() co...	matMul: OPBFOp.cc, stl_vecto...
98.35	9.55	40	MP::execute(unsigned long)	matMul: MP.cc, stl_list.h, Core...
29.93	6.62	1 044 480	void Core::Ld<float, float, (...	matMul: Core.ii, stl_vector.h, ...
16.46	6.14	2 263 040	OPBFOp::size() const	matMul: OPBFOp.h
6.62	3.75	1 546 240	CoreExtractor<float, (oper...	matMul: CoreExtractor.ii, stl_v...
6.62	3.75	1 546 240	CoreExtractor<float, (oper...	matMul: CoreExtractor.ii, stl_v...
4.20	3.04	1 167 360	MP::translate(unsigned lon...	matMul: MP.cc
14.58	2.94	491 520	void Core::Mad<float, (oper...	matMul: Core.ii, Core.h
1.72	1.72	1 003 520	unsigned long OPBFOp::im...	matMul: OPBFOp.h, stl_vector.h
3.56	1.18	61 440	Core::Barrier(Core*, OPBFO...	matMul: Core.cc, stl_tree.h, s...
1.16	1.16	122 883	Card::translate(unsigned lo...	matMul: Card.cc, stl_tree.h
1.93	1.09	450 560	CoreExtractor<int, (operan...	matMul: CoreExtractor.ii, stl_v...
0.98	0.98	1 372 206	OPBFOp::immediateSize() c...	matMul: OPBFOp.h
1.58	0.89	368 640	CoreExtractor<unsigned in...	matMul: CoreExtractor.ii, stl_v...
1.45	0.82	337 920	CoreExtractor<unsigned in...	matMul: CoreExtractor.ii, stl_v...
1.02	0.80	70 823	std::_Rb_tree_insert_and_r...	libstdc++.so.6.0.12
0.73	0.70	81 616	_int_malloc	libc-2.9.so: malloc.c, arena.c
1.23	0.70	286 720	CoreExtractor<int, (operan...	matMul: CoreExtractor.ii, stl_v...
0.78	0.68	81 616	_int_free	libc-2.9.so: malloc.c, arena.c
0.58	0.58	1	computeGold	matMul: matrixMul_gold.cpp
1.94	0.54	163 840	void Core::getValues<int, (...	matMul: Core.h
1.30	0.54	81 607	malloc	libc-2.9.so: malloc.c
1.30	0.50	81 687	free	libc-2.9.so: malloc.c
3.28	0.49	122 880	void Core::Add<unsigned i...	matMul: Core.ii, Core.h
1.97	0.46	71 680	void Core::St<float, (opera...	matMul: Core.ii, stl_vector.h, ...
2.73	0.41	102 400	void Core::Add<int, (opera...	matMul: Core.ii, Core.h
1.45	0.40	122 880	void Core::getValues<unsig...	matMul: Core.h
1.45	0.39	61 440	void Core::Mul<unsigned in...	matMul: Core.ii, Core.h, Core...

Figure 4.7: *Profiling Results for matMul*

The *kcachegrind* tool was used to analyze the profiling data generated by the *callgrind* tool. Besides generating a call graph, this application is also capable of showing a *flat profile* of all called functions, ordering them by cost (execution time). Figure 4.7 shows the initial part of the *flat profile* for the *matMul* benchmark. The *self* column refers to the total cost of the function itself, while the *incl* column refers to the cost including all called functions.

By using this information, several bottlenecks can be identified and other conclusions can be taken.

- The `Core::execute()` function, which is the main loop of execution for a core, constitutes the most significant bottleneck. The function is very small, with only 8 lines of code, but due to the fact that it is called for every instruction of every core, its impact on the performance is important.

- The `Core::translate()` function is called every time an *device address* has to be translated to a *host address*. This occurs an huge number of times: 7 million times in the *matMul* benchmark, according to collected data. The translation is basically implemented as a series of interval checks on the *device address*, to identify the memory space. Then, the address is looked up in an `std::map`, which stores the address pairs.

- The `OPBFOp::operandsSize()` and `OPBFOp::size()` functions, are called many times because of a call to the `OPBFOp::size()` function in the statement `core->incPC(op.size())`. This is called to increment the *program counter* value after each instruction. Since the size of an instruction is static, it could be calculated on creation and stored, instead of calculating it every time.

- Calls to the opcode implementation functions are not a source of significant bottlenecks. In this case, the call to the `Core::Ld` opcode implementation is the most significant one, occupying a mere 6.6% of the execution time.

Due to the general nature of these bottlenecks, they occur in all benchmarks, although not exactly in the same proportion. These are the program functions that would benefit the most of optimizing them.

Chapter 5

Conclusions and Future Work

All of the objectives of the present thesis have been met. Particularly, the development of an emulation library that parses PTX intermediate assembly language generated from a CUDA application, generating a new binary object and using it to emulate the functionality of a GPU. The library, called *libecuda*, successfully emulates the behavior of a GPU for the tested benchmarks, and should work for other CUDA applications without major changes.

Developing and debugging an emulator is a challenging task. Managing to get the emulator to run without causing *segmentation faults* was relatively straight forward, as identifying the causing problem was as easy as launching `gdb`. The hardest part proved to be identifying problems that caused the emulated programs to return an invalid result without crashing the emulator. In this case, the less time consuming way was to use the *device emulation mode*, provided by NVIDIA, to our advantage. This mode allows `printf()` calls in the kernel code, which would make no sense in the normal mode. By printing the thread ID and a variable value and then identifying which register was assigned to the variable, the expected value of a register at a certain point in time could be known. This simplified the debugging process, which would otherwise be near to impossible.

This thesis is based on the knowledge previously acquired in the *Introduction to Computer Systems* (C programming language), *Computer Architecture and Operating Systems I* (x86 assembly language and system architecture), *Computer Architecture and Operating Systems II* (operating system programming) compulsory courses and the *Advanced Programming for Telecommunications Engineering* (Java programming) elective course. During the thesis, acquired knowledge includes C++ programming and advanced techniques, emulation techniques, compiler theory and techniques,

parsing techniques, and performance profiling.

There is room for several changes to the emulation code that would reduce its currently enormous slowdown. PTX code analysis for dead code removal could be applied, as the code generated by the *nvcc* tool includes superfluous instructions. A more sophisticated register allocation algorithm could be implemented, which would reduce the register usage, which is currently about 70% larger than the code generated by NVIDIA for its Cubin format. Several of the current bottlenecks listed in Section 4.5 could be addressed, with the proposed solutions. Thread parallelization performance could increase, as it does not reach the peak value in all benchmarks. This could be done by allocating thread blocks to multiprocessors dynamically instead of statically at the start, which would eliminate the current *unbalanced execution* syndrome. One of the best optimizations that could be applied would be to substitute the current *instruction* cache with a *basic block* cache, which would reduce the number of lookups necessary. The hardest optimization, which would require a dramatic rewrite, would be to substitute the currently employed *interpretation* emulation technique with a *binary translation* technique.

Bibliography

[1] Intel Corporation. A platform 2015 workload model recognition, mining and synthesis moves computers to the era of tera, 2005. [cited at p. 3]

[2] Amar Shan. Heterogeneous processing: a strategy for augmenting moore's law. *Linux Journal*, January 2nd, 2006. [cited at p. 3, 4]

[3] Sanjay Patel and Wen-mei W. Hwu. Accelerator architectures. *Micro, IEEE*, 28(4):4–12, July-Aug. 2008. [cited at p. 4]

[4] Roger D. Chamberlain, Joseph M. Lancaster, and Ron K. Cytron. Visions for application development on hybrid computing systems. *Parallel Comput.*, 34(4-5):201–216, 2008. `http://rssi.ncsa.illinois.edu/proceedings/posters/rssi07_12_poster.pdf`. [cited at p. 4]

[5] Erich Strohmaier et al. Hans Meuer. Top 500. `http://www.top500.org/static/lists/2009/06/TOP500_200906_Poster.pdf`, June 2009. [cited at p. 4]

[6] Ken Koch. Roadrunner platform overview, March 2008. [cited at p. 5]

[7] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM. `http://bebop.cs.berkeley.edu/pubs/williams2006-cell-scicomp.pdf`. [cited at p. 5]

[8] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu. Program optimization space pruning for a multithreaded gpu. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, New York, NY, USA, 2008. ACM. `http://impact.crhc.illinois.edu/ftp/conference/cgo-08-ryoo.pdf`. [cited at p. 6, 17, 49]

[9] NVIDIA Corporation. PTX ISA version 1.3, October 2008. [cited at p. 6, 24]

[10] Wladimir J. van der Laan. Cubin utilities. `http://www.cs.rug.nl/~wladimir/decuda/`, 2007. [cited at p. 6]

[11] Jon Peddie Research. Computer graphics chip shipments dive in q4 '08 according to jon peddie research. `http://jonpeddie.com/press-releases/details/computer_graphics_chip_shipments_dive_in_q4_08_according_to_jon_peddie_rese/`, January 2009. [cited at p. 9]

[12] Wen mei Hwu and David Kirk. Ece 498 al programming massively parallel processors textbook, 2006-2008. [cited at p. 10, 12, 14]

[13] Scott Wasson. Ati dives into steam computing. http://techreport.com/articles.x/10956/1, October 2006. [cited at p. 11]

[14] Vincent Chang. Steaming into the future. http://www.hardwarezone.com/articles/view.php?cid=3&id=2821, March 2009. [cited at p. 11]

[15] NVIDIA Corporation. CUDA programming guide 2.0. July 2008. [cited at p. 12, 14, 16, 47, 54]

[16] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *ISCA*, pages 140–151, 2009. https://netfiles.uiuc.edu/jkelm2/www/papers/kelm-isca2009.pdf. [cited at p. 13]

[17] Mauricio J. Serrano, Wayne Yamamoto, Roger C. Wood, and Mario Nemirovsky. A model for performance estimation in a multistreamed superscalar processor. In *Proceedings of the 7th international conference on Computer performance evaluation : modelling techniques and tools*, pages 213–230, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc. [cited at p. 15]

[18] Victor Moya. Study of the techniques for emulation programming. Master's thesis, Facultat d'Informàtica de Barcelona, Universitat Politècnica de Catalunya, June 2001. http://personals.ac.upc.edu/vmoya/docs/emuprog.pdf. [cited at p. 17, 18]

[19] Wikipedia. Binary translation, 2009. http://en.wikipedia.org/w/index.php?title=Binary_translation&oldid=297418679. [cited at p. 17]

[20] Wikipedia. Virtual machine, 2009. http://en.wikipedia.org/w/index.php?title=Virtual_machine&oldid=300097377. [cited at p. 17, 18]

[21] Wikipedia. Dynamic recompilation, 2009. http://en.wikipedia.org/w/index.php?title=Dynamic_recompilation&oldid=285654967. [cited at p. 17]

[22] Fabrice Bellard. Qemu: Open source processor emulator. http://www.qemu.org/. [cited at p. 18]

[23] Kevin Lawton. Bochs. http://bochs.sourceforge.net/. [cited at p. 19]

[24] Sebastian Biallas. Pearpc. http://pearpc.sourceforge.net. [cited at p. 19]

[25] VMWare Inc. Vmware. http://www.vmware.com. [cited at p. 19]

[26] Sun Microsystems. Virtualbox. http://www.virtualbox.org. [cited at p. 19]

[27] Jack Veenstra. Mint simulator. http://www.wotug.org/parallel/simulation/architectures/mint/. [cited at p. 20]

[28] Wikipedia. Software performance analysis, 2009. http://en.wikipedia.org/w/index.php?title=Software_performance_analysis&oldid=300765281. [cited at p. 20]

[29] Julian Seward et al. Valgrind. http://valgrind.org/. [cited at p. 20]

[30] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009)*, pages 163–174, April 2009. http://www.ece.ubc.ca/~aamodt/papers/gpgpusim.ispass09.pdf. [cited at p. 22]

[31] David Parello Sylvain Collange, David Defour. Barra, a modular functional gpu simulator for gpgpu. Technical report, Université de Perpignan, 2009. http://hal.archives-ouvertes.fr/hal-00359342. [cited at p. 22]

[32] Gregory Diamos. Ocelot: A binary translation framework for ptx. http://code.google.com/p/gpuocelot/, July 2009. [cited at p. 22]

[33] NVIDIA Corporation. The CUDA compiler driver NVCC. http://sbel.wisc.edu/Courses/ME964/2008/Documents/nvccCompilerInfo.pdf. [cited at p. 23]

[34] Doug Brown John Levine, Tony Mason. *Lex & Yacc*. O'Reilly, 2nd edition edition, 1992. [cited at p. 25]

[35] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997. [cited at p. 31, 32]

[36] Jens Palsberg. Register allocation via coloring of chordal graphs. In *CATS '07: Proceedings of the thirteenth Australasian symposium on Theory of computing*, pages 3–3, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc. http://compilers.cs.ucla.edu/fernando/publications/papers/APLAS05.pdf. [cited at p. 31, 32]

[37] Frank Pfenning. 15-411 compiler design lecture notes on register allocation. http://www.cs.cmu.edu/~fp/courses/15411-f08/lectures/03-regalloc.pdf, September 2008. [cited at p. 31, 32]

[38] Frank Pfenning. 15-411 compiler design lecture notes on liveness analysis. http://www.cs.cmu.edu/~fp/courses/15411-f08/lectures/04-liveness.pdf, September 2008. [cited at p. 32, 33]

[39] Wikipedia. Cpu cache, 2009. http://en.wikipedia.org/w/index.php?title=CPU_cache&oldid=301440796. [cited at p. 41]

[40] Stephen Prata. *C++ Primer Plus*. Sams Publishing, 5th edition edition, November 2004. [cited at p. 41]

[41] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM. http://impact.crhc.illinois.edu/ftp/conference/ppopp-08-ryoo.pdf. [cited at p. 49]

List of Figures

List of Tables

List of Code Listings