Università di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

On expressing different concurrency paradigms on virtual execution environment

Cristian Dittamo

Supervisor

Dr. Antonio Cisternino

November 22, 2010

Abstract

Virtual execution environments (VEE) such as the Java Virtual Machine (JVM) and the Microsoft Common Language Runtime (CLR) have been designed when the dominant computer architecture featured a Von-Neumann interface to programs: a single processor hiding all the complexity of parallel computations inside its design. Programs are expressed in an intermediate form that is executed by the VEE that defines an abstract computational model in which the concurrency model has been influenced by these design choices and it basically exposes the multi-threading model of the underlying operating system. Recently computer systems have introduced computational units in which concurrency is explicit and under program control. Relevant examples are the Graphical Processing Units (GPU such as Nvidia or AMD) and the Cell BE architecture which allow for explicit control of single processing units, local memories and communication channels. Unfortunately programs designed for Virtual Machines cannot access to these resources since are not available through the abstractions provided by the VEE. A major redesign of VEEs seems to be necessary in order to bridge this gap. In this thesis we study the problem of exposing non-Von Neumann computing resources within the Virtual Machine without need for a redesign of the whole execution infrastructure. In this work we express parallel computations relying on extensible meta-data and reflection to encode information. Meta-programming techniques are then used to rewrite the program into an equivalent one using the special purpose underlying architecture. We provide a case study in which this approach is applied to compiling Common Intermediate Language (CIL) methods to multi-core GPUs; we show that it is possible to access these non-standard computing resources without any change to the virtual machine design.

To my parents and my nephew.

Acknowledgments

Many people have directly or indirectly partaken in making this thesis possible. First of all, I wish to thank my supervisor Antonio Cisternino who always supported and encouraged me, leaving me the freedom of experimenting with many different topics in different area. I wish also to thank Vincenzo Gervasi and prof. Börger for their advices on ASM modeling.

For the uncountable discussions on computer science and life we had in these three years, I also wish to thank my friends and colleagues of the CVSLab: Davide, Gabriele, Marco, Nicole, Simone, Stefano P., Stefano S.

For their invaluable contribute to make better my personal life, I wish to thank my friends: Alba, Andrea, Anna, Daniele, Daniele M., Davide, Fedele, Federica, Gaspare, Giacomo, Ilaria, Luca, Nicola, Pippo, Stefania, Veronica, and Veronika.

Last but not least I wish to thank my parents, Dina and Filiberto, that always supported me in what to do most, and Letizia, my nephew, that makes it so nice all times I come back to my hometown, and make it so hard to leave again right after.

Contents

Introduction **ix**
I.1 Research problem . . . ix
I.2 Problem statement . . . xiii
I.3 Research scope . . . xiii
I.4 Research contributions . . . xvii
I.5 Organization and Reading plans . . . xvii

1 Models of execution **1**
1.1 A brief introduction to the Abstract State Machines . . . 3
1.1.1 Basic (single-agent) ASM . . . 3
1.1.2 Control State ASM . . . 4
1.1.3 ASM Multi-agents . . . 5
1.2 Models of Sequential execution . . . 5
1.2.1 RAM . . . 6
1.2.2 RAM_L: RAM ASM model with race conditions management . . . 9
1.3 Models of Parallel execution . . . 11
1.3.1 Shared memory . . . 11
1.3.2 Distributed memory . . . 15
1.4 Model of GPGPUs . . . 19
1.4.1 The computational model . . . 20
1.4.2 The memory model . . . 23
1.4.3 The architecture model . . . 25
1.4.4 ASM ground model . . . 31

2 Expressing concurrency paradigms on VEEs **35**
2.1 Virtual Execution Environment: design and capabilities . . . 35
2.2 Strongly Typed Execution Environment . . . 37
2.2.1 Machine model and state . . . 37
2.2.2 CLR compilation toolchain . . . 39
2.2.3 Just In Time compilation . . . 39
2.2.4 Meta-data . . . 40
2.2.5 Common Intermediate Language . . . 42
2.2.6 Delegate . . . 44

2.2.7 Isolation and security boundaries . . . 46
2.2.8 Communication inter-domain . . . 46
2.2.9 Interoperability via the Platform Invocation Services . . . 46
2.3 Meta-programming and Runtime code generation . . . 48
2.4 Expressing architectural features . . . 50
2.4.1 Interface . . . 50
2.4.2 Custom annotation . . . 50
2.4.3 Object system . . . 53
2.4.4 Quotation . . . 56
2.5 Our approach: types + metadata . . . 56
2.5.1 Definitions . . . 58
2.5.2 Definitions for STEEs . . . 58
2.5.3 Expressing the RAM on the CLI . . . 62
2.5.4 Expressing the PRAM on the CLI . . . 63
2.5.5 Expressing the H-PRAM on the CLI . . . 68
2.5.6 Expressing the LogP on the CLI . . . 69
2.5.7 Expressing the GPGPUs model on the CLI . . . 69
2.5.8 Formal definition of the CIL to GPGPU compiler . . . 70

3 Parallelism exploitation 73
3.1 Why is parallel programming so difficult? . . . 74
3.2 Existing approaches to parallel programming . . . 75
3.2.1 Automatic program parallelization . . . 76
3.2.2 New programming language paradigms . . . 78
3.2.3 Extending standard design . . . 81
3.2.4 GPGPU programming languages . . . 86

4 4-Centauri: a MSIL to NVIDIA PTX compiler 105
4.1 Compiling from MSIL to PTX . . . 105
4.1.1 The philosophy . . . 107
4.1.2 Two-level compilation model . . . 108
4.2 Organization of the compiler . . . 114
4.2.1 Data Structures . . . 115
4.2.2 Code Analyzer . . . 118
4.2.3 Parser . . . 120
4.2.4 Code Generator . . . 122
4.2.5 Runtime support . . . 125

5 Evaluation 127
5.1 Mandelbrot algorithm . . . 128
5.1.1 Implementation details and results . . . 128
5.2 Mersenne Twister algorithm . . . 130
5.2.1 Implementation details and results . . . 131

Conclusions **135**
C.1 Thesis summary 135
C.2 Suggestions for Future Research 136

Bibliography **137**

List of Tables

2.1 PTX and CIL types mapping. 71

4.1 Mapping of MSIL instructions and Node classes. For the sake of brevity we do not consider efficient encoding versions of MSIL instructions. For a complete list, please see the ECMA 335, partition III [1]. 118

4.2 *NVIDIALib* wrapper types. 124

5.1 Nvidia GeForce G210M specification: main features. 127

5.2 Comparison of completion times executing the Mandelbrot *Kernel* on a CPU dual-core and on a GPU. In order to evaluate the overhead introduced by **4-Centauri**, the CT is computed twice, i.e. in the second run no compilation is performed. Moreover, we compared the **4-Centauri** compiled *Kernel* with a Nvidia CUDA implementation of it. 129

5.3 Comparison of completion times executing the Mersenne-Twister *Kernel* on a CPU dual-core and on a GPU. In order to evaluate the overhead introduced by **4-Centauri**, the CT is computed twice, i.e. in the second run no compilation is performed. Moreover, we compared the **4-Centauri** compiled *Kernel* with a Nvidia CUDA implementation of it. 133

List of Figures

I.1 The performance increase of GPUs and CPUs over the last decade (using the "texels per second" metric). x
I.2 The CLR's Multiple Instructions Multiple Data computational model is different from the GPUs' SIMD one. How does a JIT compiler can map these models? . . xii
I.3 The **4-Centauri** compiler: first a IL code is parsed to find out through annotations, which part will execute on a CPU and which will execute on GPU, if available; then two layer of abstraction in code manipulation are given: one more generic used for basic class file manipulation that leverages the CLR, and one more specific for GPU intermediate code (e.g. Nvidia PTX) generation that leverages the Nvidia CUDA driver. xv

1.1 Nvidia GPU Computational model. A cooperative thread array (CTA) is a set of concurrent threads that execute the same kernel program. A grid is a set of CTAs that execute independently. 22
1.2 Nvidia Memory Hierarchy. 24
1.3 Architectural view of data transfer between CPU and GPU (with averaged observed bandwidth) 25
1.4 AMD Cypress architecture building blocks. 26
1.5 Block diagram of the Nvidia GF100 GPU 29
1.6 Nvidia Streaming Multi-processor architecture. 30

2.1 CLI Machine State Model 38
2.2 Basic CLR toolchain: from source code to its execution. 40
2.3 A platform invoke call to an unmanaged DLL function. Figure in the article "A Closer Look at Platform Invoke" of the .NET Framework Developer's Guide. . . . 47
2.4 Mapping a C# class members to the GPGPU memory hierarchy. 57

3.1 Two-level compilation approach. 100

4.1 **4-Centauri** software stack. 107
4.2 Two-level compilation model. 109
4.3 Organization of the **4-Centauri** compiler. 114
4.4 **4-Centauri** parser component. Given a MSIL code as input it builds parse tree using Node data structure. 122

4.5 **4-Centauri** generator component. Given a parse tree as input generates a PTX code. The visit is postfix. 122

5.1 Comparison of completion times executing the Mandelbrot *Kernel* on a CPU dual-core and on a GPU. In order to evaluate the overhead introduced by **4-Centauri**, the CT is computed twice, i.e. in the second run no compilation is performed. Moreover, we compared the **4-Centauri** compiled *Kernel* with a Nvidia CUDA implementation of it.k 130

5.2 Comparison of completion times executing the Mersenne-Twister *Kernel* on a CPU dual-core and on a GPU. In order to evaluate the overhead introduced by **4-Centauri**, the CT is computed twice, i.e. in the second run no compilation is performed. Moreover, we compared the **4-Centauri** compiled *Kernel* with a Nvidia CUDA implementation of it. 134

Introduction

I.1 Research problem

Over the last decade the performance of computation devices has steadily grown. In general-purpose processors (CPUs), the number of cores[1] in a processor has increased rapidly following Moore's Law as applied to traditional microprocessors. The multi-core paradigm has become the primary method for providing performance improvements. However, sequential performance is limited by three factors or "walls": power, memory and Instruction Level Parallelism (ILP). Power is a "wall" because heat dissipation has reached a physical limit. It gets worse as gates get smaller, meaning a significant increase in clock speed without expensive cooling is not possible without a breakthrough in the technology of materials [2]. Memory is a "wall" because memory performance improvement lags increasingly behind the processor one [3]. ILP is a limit because sequential performance acceleration using ILP has stalled due to two reasons: the success of speculative execution is difficult to predict, and ILP causes an increase in power consumption without linear speedup in application performance. Patterson [4] expressed the relationship between sequential performance and these limitations with the following formula

$$\text{brick wall for sequential performance} = \text{power wall} + \text{memory wall} + \text{ILP wall}$$

The solution adopted by major CPU vendors places multiple cores onto a single die to exploit TLP[2]. These issues have encouraged researchers to explore other execution models that match the intrinsic constraints of the underlying VLSI[3] technology and the parallelism in emerging applications [5].
One result of this exploration is an increasing interest in special-purpose architectures, e.g. graphics hardware (GPUs) [6] and `Cell BE` [7], and in hybrid solutions e.g. the Intel `Larrabee` [8] now called Knights Corner.

The need for efficient real-time specialized processing of 3D meshes silently introduced an architecture model designed for computer games in ordinary PCs, called

[1]Cores are multiple copies of a processor that are placed onto a single die to exploit Thread Level of Parallelism (TLP).
[2]Thread Level Parallelism
[3]Very Large Scale Integration

GPU. It breaks with conventional Von-Neumann architectures and programming models, because it explicitly exposes parallel programming interfaces in a general purpose system in the form of a Single Instruction Multiple Data (SIMD) processor.

Commodity graphics hardware has evolved tremendously over the last years. It started with basic polygon rendering via 3dfx's Voodoo Graphics in 1996, and continued with custom vertex manipulation four years later. The GPU has improved to a full-grown graphics-driven processing architecture with a speed-performance approx.750 times higher than a decade before (1996:50mtex/s, 2006:36,8btex/s). This makes the GPU evolve much faster than the CPU, which became approximately 50 times faster in the same period (1996:66 SPECfp2000, 2006:3010 SPECfp2000) [9]. Figure I.1 shows the GPU performance over the past ten years and how the gap between CPU and GPU performances grows wider. Experts believe that this evolution will continue for the next five years at least.

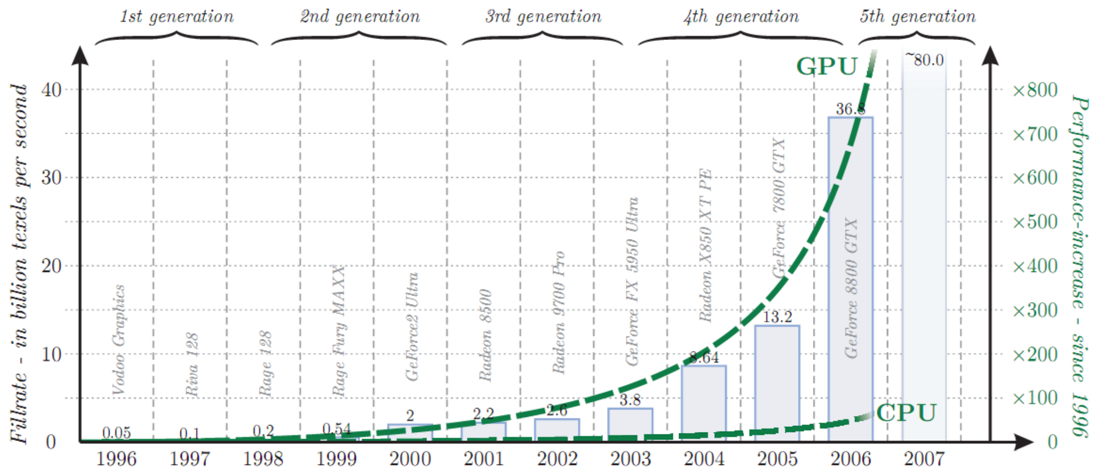

Figure I.1: The performance increase of GPUs and CPUs over the last decade (using the "texels per second" metric).

As the name implies, the GPU was initially designed for accelerating graphical tasks. For instance, the real-time 3D visual effects, e.g. interactive cinematic lighting system [10, 11], demanded by games, cinematography and other interactive 3D applications, require efficient hardware-based rendering [12].

However, GPUs were soon being exploited for performing non-graphical computations, for instance, the work of Lengyel et al. uses GPUs to compute robot motion [13], the CypherFlow project by Kedem and Ishihara exploits GPU to decipher encrypted data [14], and a GPU-based computation of Voronoi diagrams has been presented by Hoff et al. [15]. Nevertheless, most of the general-purpose algorithms implemented for GPUs stay in the academic field and have not yet found their way into industrial software engineering. The primary reason is that GPU-based application development is much more complex, mainly because the developer has to be an expert in two domains: the application domain, and computer graphics.

This approach has been influenced by the entertainment and special-effects industry, where software developers create the so-called “rendering engine” and graphics artists use the engine to create the so-called “shader programs” that compute visual phenomena. Most of the existing GPU-based development systems are founded on this programming model, e.g. RenderMan [16], Interact-SL [17], and Cg [18].

However, these development systems are ill suited for expressing general purpose computations. Being domain specific languages, they require the programmer to perform morphism from his own problem space to the CG problem space. This requires to encode the input in a CG friendly way and to decode the output of a program that must be equivalent to that needed in the original problem space.

Furthermore, because CPU and GPU-based code is developed in different programming languages, additional binding code is required to “glue” the different functionalities together. This means that changing the graphics-oriented programming model and corresponding GPU development tools may significantly reduce development complexity. For this reason since 2005 the two major GPU vendors, Nvidia and AMD, have proposed fully programmable processing units, called General Purpose GPU (GPGPU), that support non-graphics development frameworks, such as Nvidia CUDA [19], AMD CAL [20], and AMD Brook+ [21]. However, they still separate between CPU and GPU-based code.

The increasing flexibility of GPUs has enabled many applications outside the original narrow tasks for which GPUs were originally designed, but many applications still exist for which GPUs are not (and likely never will be) well suited. Word processing, for example, is a classic example of a “pointer chasing” application, dominated by memory communication and is difficult to parallelize.

This presents a set of challenges in algorithm design. One problem facing the designers of parallel and distributed systems is how to simplify the writing of programs for these systems and to augment the adaptivity of software to current, as well as, future architectures. Proposals range from automatic program transformation systems [22, 23, 24, 25, 26] that extract parallelism from sequential programs, to the use of side-effect-free languages [27, 28, 29, 30], to the use of languages and systems where the programmer must explicitly manage all aspects of communication, synchronization, and parallelism [20, 19, 31]. The problem with fully automatic schemes is that they are best suited for detecting fine grain parallelism because programmers do not fully describe the semantics of a domain, depriving the translator of high level optimizations, besides explicit parallel programming may complicate the programming task.

The need for flexibility is not related to GPU programming only but also for execution environments. In particular, in the last few years research interest has been progressively increasing in Virtual Execution Environments (VEEs), such as Java Virtual Machine (JVM) [32] and .NET Common Language (CLR) [33]. VEEs are appreciated for many reasons, such as program portability across different archi-

tectures, the ability to monitor program execution, which has proven important to enforce security aspects, the tailoring of execution onto specific architectures, and certain capabilities such as dynamic loading and reflection, which allow programs to adapt their execution depending on several environment factors, including the underlying computing architecture.

VEEs abstract aspects of the host execution environment by interposing a layer that mediates program execution through dynamic examination and translation of the program's instructions before its execution on the host CPU, as illustrated on Figure I.2. In VEEs' design was inspired by mid-90's architectures, single CPU with Von Neumann model dominating the scene when they emerged, making it harder to exploit the computing power made available by new special-purpose processors, such as GPUs and Cell BE, in a seamless way.

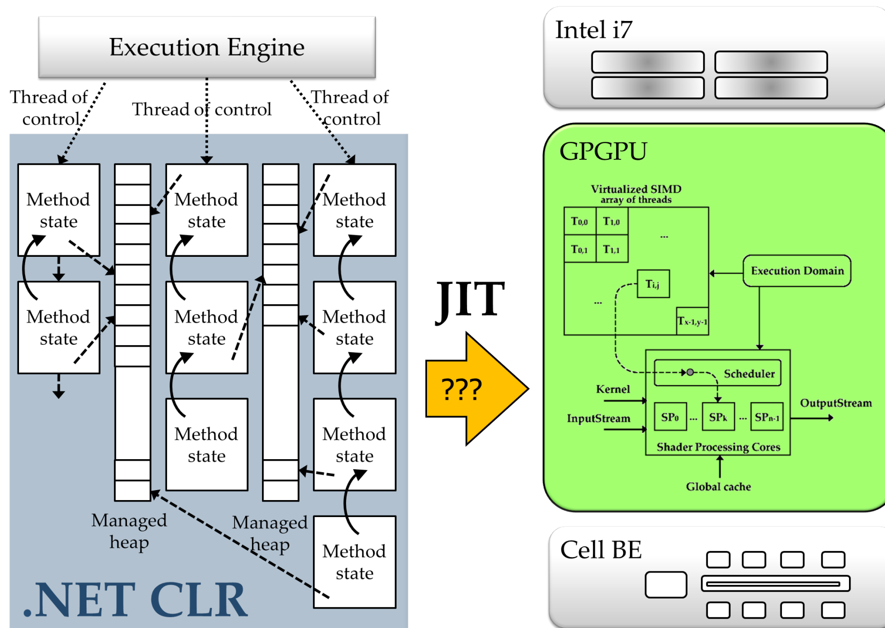

Figure I.2: The CLR's Multiple Instructions Multiple Data computational model is different from the GPUs' SIMD one. How does a JIT compiler can map these models?

In fact, the Just-In-Time (JIT) compiler module of a virtual machine can hardly exploit the power of these non-conventional processing units, since the program is expressed in an intermediate language (IL) for an abstract computation system not providing explicit representation for them. Nevertheless, there are ways to access these new computing processors by exposing specific services that bypass the VEE services.

The diverging gap between the abstract definition of the computing system and the actual architecture may significantly affect the ability of VEE programs to really

exploit the computing power of modern systems, possibly undermining the whole approach eventually abandoning it altogether and losing the many benefits that come with these VEEs. An alternative may be an acceptance that the real scope of these virtual execution environments cannot eventually avoid the need for an underlying layer of software, that coordinates pre-written software exploiting these capabilities through VEE interoperability services, as it was for Visual Basic programmers when their world was limited to the available set of C++ based COM components.

I.2 Problem statement

This thesis focuses on the problem of exposing non-conventional computing devices to VEE programs without changing the VEE base definition. Even though the problem may be tackled for specific computing devices such as GPUs or Cell BE, we will consider it from a broader perspective of exposing models of parallel computing into VEEs using a general and consistent approach showing that there is no need for a general redesign of these execution environments to adapt to non-conventional computing systems.

I.3 Research scope

To assist the reader, we summarize the key boundaries from the outset.

Models of parallel computation

Modern microprocessor architectures have considerably changed from the past: from the classical *Von Neumann* architecture to the current SIMD architecture of the GPU and the multi-core architecture of the Cell BE microprocessor. Each one of these microprocessors is completely different from the other in computational, memory and programming models. For these reasons, on expression of computational models at VEE level of abstraction, we considers two classes of parallel execution models based on shared and distributed memory respectively. Indeed, our work is not tailored to a specific architecture or execution model.

Parallelism exploitation

Indeed, the correct and efficient design of parallel programs requires to consider several different concerns, that are difficult to separate during program development. How the computation is performed using processing elements (such as processes and threads), and how these communicate, are non-functional aspects of a program since they do not contribute to define the result of the computation but only how it is performed.

- *Existing approaches*
  Several complementary approaches may help programmers achieve the construction of parallel applications.

  A way to deploy concurrency without need of change in the programming model encapsulate concurrency together with domain knowledge in common reusable library components [34, 35, 36]. For example, Tarditi et al. [37] have developed a library, named *Accelerator*, that uses data parallelism to program GPUs for general purpose uses under .NET CLR. The main advantage is that no aspect of the GPU is exposed to the programmers, only high level data-parallel operations: the programmer must use specific data(-parallel) types to program the GPU instead of the CPU. This technique can work very well, although use of multiple such libraries in the same program requires better synchronization and resource management techniques than are currently available.

  Another approach integrates concurrency and coordination into traditional languages. To build parallel applications, traditional sequential languages are extended with new features to allow programmers to explicitly guide program decomposition into parallel subtasks, as well as provide atomicity and isolation when those subtasks interact with shared data structures [38, 31, 39, 29, 19, 20]. For instance, IBM X10 [29] extends the Java sequential core language with non-functional aspects (e.g. *place*, *activities*, *clocks*, (distributed, multi-dimensional) arrays, etc.) to expose a "virtual shared-memory multiprocessor" computational model.

  A different technique raises semantic level to eliminate explicit sequencing. Parallelism can be more effectively exploited by avoiding procedural languages and using domain-specific systems based on rules or constraints [27, 21, 40, 41, 28]. Programming styles that are more declarative specify intent rather than sequencing of primitives and thus inherently permit parallel implementations that leverage the concurrency and transaction mechanisms of the system. For instance, the RapidMind Development Platform [42] allows developers to use standard C++ to create concurrent/parallel applications that run on GPUs, the Cell BE, and multi-core CPUs. It provides a single unified programming model, adding types and procedures to standard ISO C++ and relying on a dynamic compiler and run-time management system for parallel processing. OpenGL [43] and DirectX10 [44] is another successful domain-specific declarative API for graphics workloads. Many more systems have been proposed in the long history of parallel programming and we will discuss those more relevant to our work throughout the dissertation.

- *Our approach*
  To efficiently develop general-purpose applications that are accelerated by the special-purpose architectures, a different approach is needed with respect to

those emerging for specific systems such as Nvidia CUDA and AMD CAL on GPUs: the software developer creates the complete "software" that contains code for the main and special-purpose processors at the same time. This means that there is a single development environment, where the same programming language is used to define CPU and, for instance, GPU-based code side by side, and no binding code is required that connects the variables of the different processor platforms.

Subscribing the "separation of concerns" concept, typical of Aspect Oriented Programming (AOP) [45], we recognize the importance in our solution of using proper tools to program the non-functional aspects related to parallelism exploitation, e.g. optimizations, communication and synchronization management, etc. We propose a set of types and meta-data that can be used by programmers to consciously "suggest" how a parallel application can be automatically derived from the code. This is different from the standard AOP approach where join points are defined using patterns, making the programmer unaware of program transformation details that will be applied afterwards. Our underlying assumption is that the programmer can make better granularity and partitioning decisions, while the compiler can correctly manage synchronization. Since those types and meta-data express the main features of a parallel execution model, our meta-program, called **4-Centauri**, can derive how to compile source code to exploit a special-purpose architecture capabilities, as shown in Figure I.3.

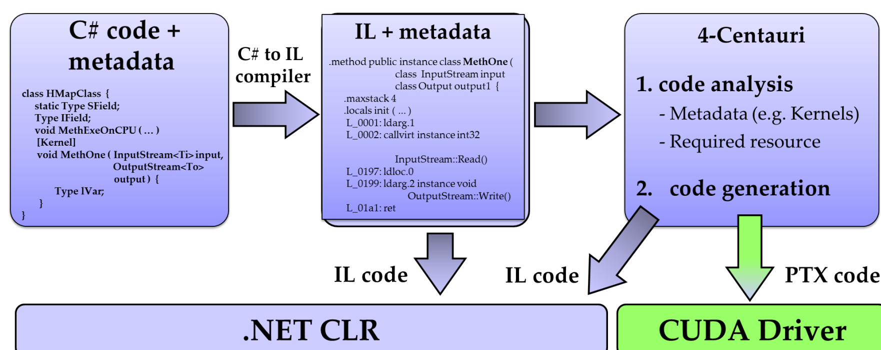

Figure I.3: The **4-Centauri** compiler: first a IL code is parsed to find out through annotations, which part will execute on a CPU and which will execute on GPU, if available; then two layer of abstraction in code manipulation are given: one more generic used for basic class file manipulation that leverages the CLR, and one more specific for GPU intermediate code (e.g. Nvidia PTX) generation that leverages the Nvidia CUDA driver.

This way it is possible to preserve the illusion of the system directly executing the program as the programmer wrote it, with no user-visible optimizations.

This constraint has several consequences that distinguish our work from other optimizing compiler implementations:

- Programmers are free to edit any procedure at the CLI level, keeping in mind which is the execution model of their implementation.
- Programmers are able to understand the execution of the program and any errors in it solely in terms of the source code and the source language constructs. This requirement on the debugging and monitoring interface to the system disallows any internal optimizations that would shatter the illusion of the implementation directly executing the source program as written. Programmers should be unaware of how their programs get compiled or optimized.
- The programmer should be isolated even from the mere fact that the programs are getting compiled at all. No explicit commands to compile a method or program should ever be given, even after programming changes. The programmer just runs the program.

4-Centauri is a compiler prototype developed as proof-of-concept. Since it is a prototype, we limit our consideration of computational models to the special-purpose one provided by GPUs.

Virtual Execution Environment

Although VMs in the form of *abstract machines* have been around for a long time (since the mid-1960s) [46, 47], the advent of Java has made them a common technique for implementing new languages, particularly those intended for use in heterogeneous environments. One of the reasons of this success is due to the (extensible) *reflection model* provided by VEEs. They enable dynamic access to the representation of the application, and allow the program to change its behavior while running depending on its current execution state [48, 49]. This dissertation describes an extension to a particular class of run-time support that we call Strongly Typed Execution Environment (STEE) whose relevant members are Sun JVM and Microsoft CLR.

Strongly Typed Execution Environment (Model). A STEE provides an extensible type system and reflection capabilities. It guarantees both that types can be established at run-time, and values are only accessed by using the operators defined on them. STEE has information about running programs, such as the state and liveness of local variables in a method; all extant objects and object references, including reachability information. Examples of STEE are the Common Language Infrastructure (CLI), the Common Language (CLR) and the Java Virtual Machine (JVM).
We limit our consideration of STEE to CLI because:

- CLI is the standard ECMA 335 [1].
- CLI is language neutral and enables multi-language applications to be developed. To assist this interoperability, a Common Type System (CTS) [1] is defined together with a Common Language Specification (CLS) [1] a set of rules that each component of the application must obey.
- there is a world-wide community of developers that uses CLI, in particular its implementations CLR, under Windows, and Mono, under Linux systems.

I.4 Research contributions

The main contributions of this thesis are:

- We show how to express the main features of the above mentioned models at VEE level without making any changes neither to the VEE design nor to its modules (e.g. JIT).
- We provide a feasible mapping between main models of parallel computations and that exposed by the VEE such as CLI, so that this work is not tailored to neither a specific architecture nor a single execution model.
- We introduce a meta-program, named **4-Centauri**, that translates a Common Intermediate Language (CIL) [1] program into a modern, mass-market, special-purpose architecture intermediate language program, such as Nvidia PTX. The goal is to provide a single and unified programming model without losing expressiveness, forcing the use of a single source language or changing the design of the VM itself.

Since we are interested in defining suitable mappings between abstract parallel computing models and VEEs resources we give a formal specification for each fundamental model of parallel computation useful to understand if and how it can be exposed into the execution environment in a consistent way. These specifications will be well-suited to study their main features, by using the Abstract State Machines (ASM) framework.

I.5 Organization and Reading plans

The thesis is structured as follows:

- Chapter 1 surveys the main models of parallel execution highlighting their computational, memory, and execution aspects that are most relevant for our

purposes. For each of them, a formal specification is given by using the Abstract State Machine (ASM) method, a mathematically well-founded framework for system design and analysis. To make this thesis self-contained, a basic introduction to ASM is provided in this chapter.

- Chapter 2 provides a basic introduction to the Virtual Execution Environment (VEE) design and capabilities, in particular of a special type of VEE, named Strongly Typed Execution Environment (STEE). Therefore, it reviews existing approaches for exposing underlying architecture features to STEEs level. It introduces our approach based on types and meta-data, and, for each model of parallel execution, presents a mapping on the CLI leveraging ASM model defined in the previous chapter. In particular, it focuses on GPGPUs mapping on CLR, providing a formal definition and a proof of correctness of it.

- Chapter 3 reviews different approaches to parallel programming that have been studied and have inspired our solution. In particular, our discussion focuses on GPGPUs existing main programming languages: Nvidia CUDA, AMD CAL, Nvidia PTX and AMD IL.

- Chapter 4 introduces our implementation of the mapping between GPGPUs and CLR, named **4-Centauri**. This meta-programming tool is able to translate Common Intermediate Language (CIL) to Nvidia PTX language leveraging special meta-data provided by programmers. A detailed description of **4-Centauri** design and main features are provided. At the end of this chapter, some examples of compilation are presented.

- Chapter 5 provides the evaluation of the **4-Centauri** compiler by using a classical algorithm in graphics.

- Chapter 6 presents the conclusions of this thesis and discusses future work.

Chapter 1

Models of execution

While new hardware architectures offer much more computing power, they make writing software that can fully benefit from the hardware much harder. In scientific applications, improved performance has historically been achieved by having highly trained specialists modify existing programs to run efficiently as new hardware became available. However, rewriting programs is far too costly, so most organizations focused on rewriting small portions only of the mission critical programs called *Kernels*. In the optimal case, mission-critical applications spent 80%-90% of their execution time in these *Kernels*, which represent a small percentage of the application code. This rewriting was time consuming, and organizations had to balance the risk of introducing subtle bugs into well-tested programs against the benefit of increased speed at every significant hardware upgrade. All bets were off if the organization did not have the source code for the critical components. In contrast, commercial vendors have become accustomed to a world where all existing programs get faster with each new hardware generation. Software developers could confidently build new innovative software that barely run on the then current hardware, knowing that it would run quite well on the next generation machine at the same cost. This will no longer occur for sequential codes, but the goal of new software development tools must be to retain this very desirable characteristic as we move into the era of many-core computing. Designers of parallel algorithms for modern computers have to face at least four sources of considerable challenges. First, the underlying computational substrate is much more intricate than it is for conventional sequential computing, thus the design effort is much more onerous. Second, the resulting algorithms have to compete with and outperform existing sequential algorithms that are often better understood and highly optimized. Third, the ultimate reward of all this effort is limited, at best a speedup of a constant factor: the number of processors. Fourth, machines differ, and speedups obtained on one machine may not translate to speedups on others, resulting in a design effort that may be substantially wasted. A good computational model can simplify the complicated work of the software architect, algorithm designer and program developer while mapping their work effectively onto real computers. Such a computational model is sometimes also called

Bridging model [50]. The bridging model between the sequential computer and algorithm designer/program developer is the Von Neumann [51] and RAM (Random Access Machine) Model [52]. However, no commonly recognized bridging models are found between parallel computer and parallel programs, and no other model exists that can map a user's parallel program so smoothly onto parallel computers as the Von Neumann and RAM Model do. This situation is largely due to the immature parallel computer design that should take into account many different architectures for parallel computers that change rapidly each year.

Skillicorn presents three requirements for a model of parallel computation [53]:

- architecture independence, a model is general enough to represent a range of architecture types;
- congruence over an architecture, the real costs of execution on that architecture are reflected at the model level;
- intellectual manageability, a model must abstract away from the task of specifying and managing the parallelism.

There are several levels at which a model of parallel computation may exist, that are classified by McColl [54].

In the following sections, we examine models principally identified as *cost model* in the McColl's classification. For each model we developed a formal representation of its datapath. Our ground model is uniform to architectural features such as the register-file size, the datapath width, the instruction set, etc. We focus on describing how different models of parallel computation access memory. Therefore, we abstract all the instructions that are not related to memory management. We consider that the computational task of a processor is specified as an ordered sequence of instructions. The execution of the instructions follows a state transition semantics, therefore the models described in following sections behave like a finite state machine. Each execution phase is reflected in our ground models by control states. For these reasons our formal representations are based on the Abstract State Machines (ASM) method.

As proven in [55, 56, 57], ASMs have introduced a software design and analysis method which provides the right level of abstraction to capture the essential characteristics of existing architectural models and to describe them in a simple, uniform manner that clearly defines functionalities.

The method bridges the gap between non-HPC[1] programmers understanding and formulation of real-world architectures and the deployment of their algorithmic solutions by code-executing machines on changing platforms (i.e. special-purpose

[1] High Performance Computing

architectures). It covers within a single conceptual framework both design and analysis, for procedural single-agent (e.g. single processor CPU) and for asynchronous multiple-agent systems (e.g. multi-core CPU and many-core GPUs).

1.1 A brief introduction to the Abstract State Machines

ASMs represent a mathematically well founded framework for system design and analysis introduced by Gurevich as Evolving Algebras [58]. At the end of '90 Börger and Stärk in [56] have applied ASMs to the software engineering allowing to build models in a faithful and objectively checkable manner. Therefore, using ASMs is possible to turn descriptions expressed in application domain (e.g. model of parallel computations) into precise abstract definitions, which we were comfortable to manipulate as a semantically well-founded form of pseudo-code over abstract data.

1.1.1 Basic (single-agent) ASM

The ASMs are defined in [55, 56]. A brief summary is presented here in order to make the thesis self-contained.
An basic (single-agent) ASM is a transition system which transforms structures of a given signature, i.e. finite sets of so called transition *rules* of form

if *Condition* **then** *Updates*

where the *Condition* is a closed predicate logic formula of the underlying signature without free variables, whose interpretation evaluates to true or false. *Updates* is a finite set of assignments of the form $f(t_1,\ldots,t_n) := t$ whose execution is to be understood as changing (or defining if there was none) in parallel the value of the occurring functions f at the indicated arguments to the indicated value. The notion of ASM *states* is the notion of mathematical structures where data come as abstract objects, i.e. as elements of sets (also called *domains* or *universes*, one for each category of data), which are equipped with basic operations and predicates.
In any given state, first all parameters t_i, t are evaluated to their values, say v_i, v, then the value of $f(v_1,\ldots,v_n)$ is updated to v which represents the value of $f(v_1,\ldots,v_n)$ in the next state. Such pairs of a function name f and an argument $(v_1,\ldots,v_n)$ are called *locations*, location-value pairs (loc,v) are called *updates*.
An ASM computation step in a given state consists in executing simultaneously all updates of all transition rules whose guard is true in the state, if these updates are consistent, in which case the result of their execution yields the next state. In the case of inconsistency the computation does not yield a next state. A set of updates is called consistent if it contains no pair of updates with the same location, i.e. no

two elements (loc,v),(loc,v') with $v \neq v'$.
Simultaneous execution is enhanced by the following notation to express the simultaneous execution of a rule R for each x satisfying a given condition φ:

forall x **with** φ
R

where φ is a Boolean-valued expression that determines which x is/are applicable, and R a rule. Typically, x will have some free occurrences in R which are bound by the respective quantifier.
Similarly non-determinism can be expressed by rules of the form

choose x **with** φ
R

where φ is a Boolean-valued expression and R a rule. The meaning of such an ASM rule is to execute rule R with an arbitrary x chosen among those satisfying the selection property φ. If there exists no such x, nothing is done. It is possible to use combinations of **where** , **let** , **if** - **then** -**else** , etc. which are easily reducible to the above basic definitions. When dealing with multi-agent systems we use sets of agents each executing its own ASM.

1.1.2 Control State ASM

In the rest of the thesis we use a special class of ASMs, called *control state ASMs*, which allows one to define machines providing the main control structure of Finite State Machines (FSMs) synchronous parallelism and the possibility to manipulate data structures. A *control state* ASM is an ASM whose rules are all of the following form:

if $ctl_state = i$ **then**
if $cond_1$ **then**
$rule_1$
$ctl_state := j_1$
...
if $cond_n$ **then**
$rule_n$
$ctl_state := j_n$

The finitely many control states $ctl_state \in 1,...,m$ resemble the so-called 'internal' states of FSMs. In a given control state i, these machines do nothing when no condition $cond_j$ is satisfied.

1.1.3 ASM Multi-agents

In [56] the single-agent ASMs are extended to two kind of multi-agent ASMs: synchronous (*sync* ASM) and asynchronous ASMs (*async* ASM), which support modularity for design of large systems. A multi-agent synchronous ASM is defined as a set of agents which execute their own basic ASMs in parallel, synchronized using an implicit global system clock. Semantically *async* ASM is equivalent to the set of all its constituent single-agent ASMs, operating in the global states over the union of the signatures of each component. The practical usefulness of *sync* ASMs derives from the possibility of equipping each agent with its own set of states and rules and of defining and analyzing the interaction between components using precise interfaces over common locations.
An asynchronous ASM is given by a family of pairs $(a, ASM(a))$ of pairwise different agents, elements of a possibly dynamic finite set AGENT, each executing its basic ASM $ASM(a)$. A run of an *async* ASM, also called a partially ordered run is a partially ordered set $(M, <)$ of moves (execution of rules) m of its agents satisfying the following conditions:

- finite history: each move has only finitely many predecessors, i.e. for each $m \in M$ the set $\{m^{'} | m^{'} < m\}$ is finite;
- sequentiality of agents: the set of moves $\{m | m \in M, a\ performs\ m\}$ of every agent $a \in Agent$ is linearly ordered by $<$;
- coherence: each finite initial segment X of $(M, <)$ has an associated state $\sigma(X)$, i.e. the result of all moves in X with m executed before $m^{'}$ if $m < m^{'}$, which for every maximal element $m \in X$ is the result of applying move m in state $\sigma(X - m)$.

The moves of the single agents can be atomic or durative, but for simplicity the preceding definition of distributed runs assumes actions to be atomic. Multi-agent ASMs provide a theoretical basis for a coherent global system view for concurrent sequential computations of single agents, each executing its own sequential ASM, at its own pace and with atomic actions applied in its own local states, including input from the environment as monitored functions.

The relation between global and local states is supported by the use of the reserved name *self* in functions and rules to denote the agents which are executing the underlying "same" but differently instantiated basic or sync ASM, similar to the use of *this* in object-oriented programming to denote the object for which the currently executed instance method has been invoked.

1.2 Models of Sequential execution

Before examining models of parallel execution, a brief introduction is made of RAM, the simplest model of sequential execution, providing the background from which

the models of parallel execution were developed. After RAM, we will present RAM_L that extends the RAM ASM model with race condition managements rules.

1.2.1 RAM

The random-access machine (RAM) models the essential features of the traditional sequential computer. The RAM is modeled by a Finite State Machine, a central processing unit, (CPU). It implements a fetch-and-execute cycle in which it alternately reads an instruction from a program stored in the random-access memory and executes it. In the RAM model there are two types of memory that differ in size and access speed: a register-file `RF` , that is a very fast memory but it has a small number of storage units, and a random-access memory, that is slower than `RF` but it has a large number of storage units. All operations are performed by the CPU on data stored in its registers.

Definition 1.2.1. *The `RF` is the single processor local memory that can be accessed by only one agent at a time.*

A CPU typically has a set of instructions, such as arithmetic and logical ones, memory load and store ones for moving data between memory locations and registers, etc. A RAM program is a finite sequence of assembly language instructions. A valid program is one for which each jump instruction goes to an existing label. We assume for simplicity of exposition and without loss of generality that the last instruction of code is HALT. Since we are not interested here in what a CPU computes during its execution, our view of use and assignment actions is simple: an assignment action changes the value of either a memory unit or a register, and a use action does nothing; for this reason we introduce the NOP macros.

ASM ground model

We start by developing a simple abstract mathematical model, called `M`, of the RAM processor architecture, which is the starting point for the stepwise formalization of other processor architectures. In our model we define the following universes:

- UNITADDR is a memory unit[2].
- VAL is the data held in either a memory unit or a register.
- REGISTER is the finite set of registers in the `RF` .
- OPCODE = {HALT, OTHER, LOAD, STORE, LOCK, UNLOCK, SEND, RECEIVE}.
- INSTRUCTION is the instruction set of a processor[3].

[2]Based on the real architecture, a memory unit can be either a location, page, or segment.
[3]In literature this set is also called Instruction Set Architecture (ISA)

- ValidInstr = {Halt, Other, Load, Store}. This is the sub-set of instructions that can be executed by the current machine M.

- Executor ⊆ Agent, a sub-set of all possible agents. Each agent, called *executor*, executes a list of instructions selected from the Instruction set.

And we introduce the following functions:

- static
  - *op* : Instruction → OpCode returns the opcode of a given instruction.
  - *memUnitAdr* : Instruction → UnitAddr returns the address of the given instruction.
  - *value* : Instruction → Val returns the value of a given instruction's operand.

- controlled
  - *regfile* : Register → Val returns the value held in the given register
  - *mem* : UnitAddr → Val returns the value held at the given address

- monitored
  - *legal* : Instruction × Executor → Boolean returns true if a given executor executes a given instruction ∈ ValidInstr at all points in time.

We define two functions for memory access because of cost difference in access time required for different types of memory.

The main rule of M provides a formal model of the RAM processor fetch-and-execute cycle:

M(e) ≡
  case *state*(e) **of**
    Fetch : *FETCH*(e)
    Execute : *EXEC*(*currinstr*(e), e)

where *currinstr* holds the current instruction that is executed by an executor e, and *state* holds the current state of M. We have defined two macros, *FETCH* and *EXEC*, one for each phase of the fetch-and-execute cycle. When *state*(e) is equal to *FETCH*, M non-deterministically chooses the next instruction to execute. The *state* is updated to *Execute* such that in the next run M will execute the *EXEC* macro. Following are the rules for both macros:

FETCH$(e) \equiv$
  choose *instr* **in** INSTRUCTION **with** $legal(instr, e)$ **in**
    $currinstr(e) := instr$
    $state(e) := Execute$

Which instruction is selected depends on the result given by the *legal* function. This function guarantees that, given a $e \in$ EXECUTOR, M(e) never computes any instruction $\notin$ VALIDINSTR.

EXEC$(instr, e) \equiv$
  case $op(instr)$ **of**
    HALT : $HALT(e)$
    STORE : $STORE(memUnitAdr(instr), val(instr), e)$
    LOAD : $LOAD(memUnitAdr(instr), reg(instr), e)$
    LOCK : $LOCK(memUnitAdr(instr), e)$
    UNLOCK : $UNLOCK(memUnitAdr(instr), e)$
    SEND : $SEND(d, value, e)$
    RECEIVE : $RECV(s, destReg, e)$
    OTHER : NOP

To move data between memory units and RF in a RAM processor, there are two memory instructions: *load* and *store*. Reading data means to access a memory unit at the address held in a register and load it in another register. Writing data means to store data in a memory unit at the address held in a given register. This is formalized in the following two rules:

LOAD$(adr, reg, e) \equiv$
  $regfile(reg, e) := mem(adr)$
  $state(e) := Fetch$

STORE$(adr, value, e) \equiv$
  $mem(adr) := value$
  $state(e) := Fetch$

In this RAM model the other macros do not change the state of M.

LOCK$(adr, e) \equiv$
  NOP

UNLOCK$(adr, e) \equiv$
  NOP

SEND$(d, value, e) \equiv$
  NOP

RECV$(s, destReg, e) \equiv$
  NOP

HALT$(e) \equiv$
  $state(e) := idle$

When M executes the HALT macro, the control state is updated to *idle*. This means that no more updates to the M' state will be performed. The assumption for the initial state of M is:

- $state$(e) := $Fetch$
- $currinstr$(e) := $undef$

1.2.2 RAM_L: RAM ASM model with race conditions management

In this section we refine the basic single-core architecture of the RAM model considering a multiple threads running on the same processor. In this scenario, multiple executors act concurrently, therefore we must introduce new rules to manage synchronization problems; i.e. conflicts on shared resources which could result when concurrent executors access the same memory units at the same time. For this reason:

1. we extend the ValidInstr universe with two instructions, Lock and UnLock, such that the new set of valid instructions is:

   $$\text{ValidInstr} = \{\text{Halt}, \text{Other}, \text{Load}, \text{Store}, \text{Lock}, \text{UnLock}\}$$

2. we define three new functions:

   - *owner* : UnitAddr $\rightarrow$ Executor
     returns the executor that has obtained a lock on a given memory unit address
   - *lockcnt* : UnitAddr $\times$ Executor $\rightarrow$ Integer
     counts the number of locks required by a given executor on a unit of memory. For *lockcnt* there is a injective function *succ* (the successor function) such that *succ*(lockcnt) is a natural number.
   - *locked* : UnitAddr $\times$ Executor $\rightarrow$ Boolean
     where
     $$locked(adr, e) = \begin{cases} true & \text{lockcnt(adr,e)} > 0 \\ false & \text{otherwise} \end{cases}$$

3. we define two new macros, *LOCK* and *UNLOCK*, that are used by executors to acquire and release locks on a given address. They are formalized in the following two rules:

LOCK$(adr, e) \equiv$
  if $owner(adr) = e$ **then**
    $lockcnt(adr, e) := lockcnt(adr, e) + 1$
    $state(e) := Fetch$
  else
    if $owner(adr) = undef$ **then**
      $owner(adr) := e$
      $lockcnt(adr, e) := 1$
      $state(e) := Fetch$

In the second *else* branch, the *state*(e) := *Execute* such that in the next run the LOCK instruction will be executed continuously until the *owner*(adr) is either equal to e or *undef*.

UNLOCK(adr,e) $\equiv$
  if $owner(adr) = e$ **then**
    if $lockcnt(adr, e) = 1$ **then**
      $owner(adr) := undef$
    $lockcnt(adr, e) := lockcnt(adr, e) - 1$
  $state(e) := Fetch$

where the dynamic function *lockcnt* the number of times an agent has to exit the LOCK before it is released, and *owner* returns the executor that has obtained the current lock.

The main rule of `M` and both the *FETCH* and *EXEC* macros remain unchanged in this model.

Data transfer from to memory

To avoid race conditions for each memory access the processor must first check whether or not an executor has already requested a lock on a given address. This is done by calling the static function *locked*:

LOAD(adr,reg,e) $\equiv$
  if $locked(adr, e)$ **then**
    $regfile(reg, e) := mem(adr)$
    $state(e) := Fetch$

STORE(adr,value,e) $\equiv$
  if $locked(adr, e)$ **then**
    $mem(adr) := value$
    $state(e) := Fetch$

In both macros, if a given *adr* is locked by another executor, the *state*(e) := *Execute*, such that in the next `M` runs the lock test will continue to be performed, until the lock on that *adr* will be released.

The assumption for the initial state of `M` is:

- *state*(e) := *Fetch*
- *currinstr*(e) := *undef*
- *owner*(adr) := *undef*, for all $adr \in$ UNITADDR
- *lockcnt*(adr,e) := 0, for all $adr \in$ UNITADDR and $e \in$ EXECUTOR

There is an important theorem to state about synchronization management.

Theorem 1.2.2. *Let $e \in$ EXECUTOR, and let $adr \in$ UNITADDR, $UNLOCK(adr, e)$ does not release a lock that e has not acquired on adr before.*

Proof. Since the $LOCK(adr, e)$ will update the control state variable *state*(e) to *Fetch*, and execute the next instruction, iff e acquires a lock on *adr*. □

1.3 Models of Parallel execution

Having introduced the main models of sequential execution, this section surveys a selection of models of parallel execution. Parallel computation models can be classified into two categories according to the memory model of their target parallel computers: shared and distributed memory models [59, 60]. For each model we highlight three fundamental aspects: memory and synchronization management, and communication cost.

1.3.1 Shared memory

PRAM

With the success of the RAM model for sequential computation, a natural way to model parallel computation is to extend this model with parallel processing capability: `P` independent processors that share one common global memory pool. Such parallel computational model, named PRAM, was first proposed by Fortune and Wyllie [61]. It consists of an unbounded number, `P`, of processors working synchronously. Execution may be in either SIMD[4] or MIMD[5] mode, thus with a single or multiple flows of control respectively.

Synchronization. The RAM processor can execute instructions concurrently in unit time and in lock-step with automatic and free synchronizations.

Memory levels. Each processor is a RAM machine with a set of registers rather than a local memory. All processors share an unbounded global memory, via which they communicate. With the only memory being the global memory, this is a one-level memory model.

Memory access. Concurrent access to the global memory is allowed and only takes one unit of time to be finished. There are several variations of the PRAM model [59] which make different assumptions on concurrent memory access arbitration mechanism. The less restricted variation is the CRCW PRAM model, which allows concurrent read and concurrent write to the same memory unit with a mechanism for arbitrating (Priority, Common and Arbitrary) simultaneous writes to the same unit. By restricting the simultaneous write to one processor, the CREW PRAM model was proposed in [61]. In such a model, simultaneous read to the same unit was still allowed while only one processor may attempt to write to a unit. The most restricted form of the PRAM model is the EREW PRAM model, which requires that no two processors can simultaneously access any given memory unit.

[4] Single Instruction, Multiple Data
[5] Multiple Instruction Multiple Data

Advantages. Algorithm designers can focus on the fundamental computational difficulties and on the logical structure of parallel computations required to solve the problem. It is easy to program because it is a simple extension of the RAM model: the instruction set is composed of the RAM one plus a global read and write. It hides issues like reliability, synchronization, message passing, and other machine-related problems. The basic work measure is the product of the time to perform the algorithm and the number of processors used. This model provides a total expressibility, since there are no constraints on the forms of computation and communication which may be performed.

Disadvantages. It does not take into account the latency and bandwidth costs of communication. There is no abstraction from managing many parallel threads and synchronization. Since the PRAM is too abstract for realistic implementation, there are several variants that take into account different aspects of parallel architecture: asynchrony (APRAM [62] and Asynchronous PRAM [63]), memory contentions management (Module Parallel Computer [64]), latency (LPRAM [65] and BPRAM [66]), and hierarchical parallelism (YPRAM [67]).

ASM ground model. The PRAM model extends the RAM_L one with two more memory access rules. PRAM considers three different memory access rules: *EREW* (as for RAM_L), *CREW* and *CRCW*. For each of these we define different *LOAD* and *STORE* macros. In the last two rules, the management of read and read/write respectively is left to programmers.

- *EREW* (Exclusive Read, Exclusive Write)

  LOAD(adr,reg,e) ≡
  if *locked*(adr, e) **then**
  regfile(reg, e) := *mem*(adr)
  state(e) := *Fetch*

  STORE(adr,value,e) ≡
  if *locked*(adr, e) **then**
  mem(adr) := $value$
  state(e) := *Fetch*

- *CREW* (Concurrent Read, Exclusive Write)

  LOAD(adr,reg,e) ≡
  regfile(reg, e) := *mem*(adr)
  state(e) := *Fetch*

  STORE(adr,value,e) ≡
  if *locked*(adr, e) **then**
  mem(adr) := $value$
  state(e) := *Fetch*

- *CRCW* (Concurrent Read, Concurrent Write)

  LOAD(adr,reg,e) ≡
  regfile(reg, e) := *mem*(adr)
  state(e) := *Fetch*

  STORE(adr,value,e) ≡
  mem(adr) := $value$
  state(e) := *Fetch*

The main rule of `M` as well as the *FETCH* and *EXEC* macros and the initial state remain unchanged in this model.

H-PRAM

The Hierarchical PRAM (H-PRAM) model, proposed by Heywood and Ranka [68], consists of a collection of individual synchronous PRAMs that operate asynchronously from one another. A *hierarchy relation* defines the organization of synchronization between independent PRAMs. Suppose we have P-processor H-PRAM. At level 1 of the hierarchy, which is the view of the model before an algorithm executes its first step, there is a single P-processor synchronous PRAM. Allowable instructions are all PRAM instructions plus one: the *partition instruction*. It splits the P processors into disjoint subsets (along with a corresponding disjoint block of shared memory that is private to it). Each subset of processors is a synchronous PRAM operating separately from the others. This is seen as level 2 of the hierarchy. When the algorithm terminates, a sub-PRAM waits until all of the algorithms on the other sub-PRAMs terminate: this is supposed to be implemented by a synchronization step on the sub-PRAMs which follows every partition step. When the synchronization step on the sub-PRAMs finishes, the next step of the P-processor PRAM algorithm at level 1 begins. Each of the algorithms in level 2 is designed with a parameterized number of processors, so from its vantage point it is at level 1 of a hierarchy. partition steps can be used here to create level 3 of the hierarchy. This process may continue recursively until partitioning is no longer possible.

Memory levels. This model exposes a multiple-level memory model, since the hierarchy of synchronous PRAMs is dynamically configurable.

Memory access. There are two variants of the H-PRAM:

1. Private H-PRAM. The partition instruction splits the shared memory proportionately along with the processors, such that each sub-PRAM has its own private contiguous block of shared memory disjoint from the shared memories of the other sub-PRAMs. This implies a multi-level hierarchical memory, where access time in a sub-PRAM to its private block of shared memory is a function of the size of the sub-PRAM, thus taking into account the increased locality within a sub-PRAM.

2. Shared H-PRAM. The shared memory is not partitioned, thus each sub-PRAM shares the global memory.

Synchronization A hierarchy relation defines the organization of synchronization between independent PRAMs and how the individual PRAM algorithms run on the sub-PRAMs. There are two types of synchronization:

- α-synchronization, occurring between the processors of a PRAM instruction step. When all processors are running a sequence of local computations it is

unnecessary for them to α-synchronize with each other until a communication is required.

- β-synchronization, occurring between the sub-PRAMs at the end of a partition step. This cost can be ignored if it will not dominate the computation plus communication cost.

Both types are functions of the numbers of processors being synchronized (i.e. sub-PRAMs size). It is important to consider that synchronization costs need not be accounted for in a model since they can be subsumed into the costs of communication operations being synchronized [69].

Communication H-PRAM is parameterized by latency. This cost is a function of the number of processors being communicated amongst, and defined by, the specific underlying architecture.

Advantages. This model accounts for and provides abstract control over communication and hierarchical parallelism, thus being more reflective of various parallel architectures and communication locality. The H-PRAM is architecture independent , despite not accounting for communication bandwidth. Moreover, it partially abstracts from the expression of parallelism, communication and synchronization through its use of the hierarchy.

Disadvantages. Despite satisfying Skillicorn's requirements more than simpler models, the cost formula can be complicated, as shown in [59].

ASM ground model. The H-PRAM model extends the PRAM model with the possibility of processors and memory partitioning. This adds a constraint on memory accessibility: a sub-PRAM accesses its partition of the global memory only. Therefore, a check on accessibility must be performed before every memory load/store. This is formalized by the following static function:

$$legalPartition : \textsc{Executor} \times \textsc{UnitAddr} \times \text{LEVEL} \rightarrow \textsc{Boolean}$$

$$legalPartition(e, adr, level) = \begin{cases} true & \text{if } e \text{ and } adr \text{ are relative to the same sub-PRAM} \\ & \text{at a given level} \\ false & \text{otherwise} \end{cases}$$

Moreover, the definition of *legal* function must be changed as follows:

$$legal : \textsc{Instruction} \times \textsc{Executor} \rightarrow \textsc{Boolean}$$

$$legal(instruction, e) = \begin{cases} true & \text{if instruction} \in \textsc{ValidInstr} \text{ and} \\ & \quad \text{legalPartition(e,memUnitAdr(instruction))} \\ false & \text{otherwise} \end{cases}$$

Both synchronization types are already formalized by functions and macros introduced in the RAM_L model. The initial state and other macros remain unchanged in this model.

1.3.2 Distributed memory

There are several parallel computational models assuming a distributed memory and communication by message passing paradigm.

Bulk-Synchronous Process

The BSP model proposed by Valiant [70] attempts to bridge theory and practice with a more radical departure from the PRAM. A BSP parallel computer requires:

- `P` processors/memories components;
- a router that delivers point-to-point messages between pairs of components;
- facilities for barrier synchronization.

Computations in BSP consist of a sequence of supersteps with periodicity L. In each superstep each component is allocated a task consisting of a combination of computation steps using data available locally before the start of the superstep, and message transmissions, which are not guaranteed to have completed until the next superstep.

Memory levels. BSP differentiates memory that is local to a processor from that which is remote, but it does not differentiate network proximity. Therefore, this is a two-level memory model, allowing the use of strict locality.

Synchronization After each period of L time units, called τ, a global check is performed to ensure all components have finished one superstep. The cost of wait steps is subsumed in the cost L of synchronizing up to P processors. In general, L is determined empirically from timings by running benchmarks.

Communication All communications are implemented as bulk synchronizations, that includes point-to-point messages. The BSP model posits bandwidth limitation on the algorithm through limiting the maximum messages that can be sent/received

in each superstep, $h = L/g$, where g is the minimum time gap between messages. The time step T_s taken for a single superstep can be expressed as follows:

$$T_s = \underset{L<i<P}{\text{Max}}(w_i) + \underset{L<i<P}{\text{Max}}(h_i * g) + L \tag{1.1}$$

where i ranges over processes, w_i is the local computation time and h_i is the number of message send/receive of process i. BSP specifies no particular topology only the general communication characteristics indicated by L and g. All processors are seen as being equally distant, so equally costly to communicate with.

Advantages. The BSP model is architecture independent and is essentially congruent. A BSP programming library, called BSPlib [71], provides the SIMD parallelism and primitives for bulk synchronous remote memory access and message passing.

Disadvantages. The BSP model has strong requirements on the capability of latency hiding in the design and implementation of a parallel algorithm. BSP model does not abstract from the expression of parallelism, nor does it abstract from synchronization.

ASM ground model. The BSP ASM model modifies the PRAM one with features related to distributed memory and bulk synchronization. In this model each executor runs on a different RAM processor. An executor can access values residing in locations either in its local or remote memory units. In the former case, the rules performed are the same as the RAM model's. In the latter case, executors exchange data values through message passing that is formalized with two new instructions: SEND and RECEIVE. The new set of valid instructions is:

$$\text{VALIDINSTR} = \{\text{HALT}, \text{OTHER}, \text{LOAD}, \text{STORE}, \text{SEND}, \text{RECEIVE}\}.$$

In order to formalize the global check at the end of a superstep, we add a new control state, called *Wait*, the *SYNC* macro, and the following monitored function:

$$globalCheckingTime = \begin{cases} true & \text{if } \tau \text{ ends} \\ false & \text{otherwise} \end{cases}$$

Therefore the new main rule of `M` is:

M$(e) \equiv$
  if $globalCheckingTime$ **then** $state(e) := Wait$
  else case $state(e)$ **of**
    $Fetch : FETCH(e)$
    $Execute : EXEC(currinstr(e), e)$
    $Wait : SYNC(e)$

This global check is constrained in time since it must be executed after each period τ of L unit times.

SYNC$(e) \equiv$
  forall *executor* **in** EXECUTOR **with** *executor* $\neq e$
    if $state(executor) = Wait$ **then**
      $state(e) := Execute$

Only when all executors are synchronized, a new superstep starts.

Neither new computation steps nor new message transmissions are performed at the end of τ. The control state is updated to *Wait* such that all executors must wait for each other in a global synchronization.
The data movement instructions must be changed in order to consider the bulk synchronization and that different actions are required whether a data is in a local or in a remote memory unit. This is formalized with the following function:

$$isLocal : \text{UNITADDR} \times \text{EXECUTOR} \rightarrow \text{BOOLEAN}$$

that returns *true* if a given address is the local memory of a given executor. Therefore, the new rules for *LOAD* and *STORE* macros are:

LOAD(adr,reg,e) $\equiv$
  if $isLocal(adr, e)$ **then**
    $regfile(reg, e) := mem(adr)$
    $state(e) := Fetch$
  else
    $RECV(owner(adr), regfile(reg, e), e)$

STORE(adr,value,e) $\equiv$
  if $isLocal(adr, e)$ **then**
    $mem(adr) := value$
    $state(e) := Fetch$
  else
    $SEND(owner(adr), value, e)$

We abstract from communication details by the introduction of the message queue, called *recvBuffer*, where executors both store (enqueue) and load (dequeue) messages. The maximum length of *recvBuffer* is h as stated by this model definition. Before sending data values, an executor must check whether the destination can receive them or not, that is, *recvBuffer* is not full. Since we consider blocking communication instructions, if *recvBuffer* is full, that executor must wait (i.e. state(e) = *Execute*) until at least one position is free.
No more communications are performed if *globalCheckingTime* is true. This is formalized with the following two macros:

RECV(s,destReg,e) $\equiv$
  if $\neg s.recvBuffer.isEmpty$ **then**
    $destReg := s.recvBuffer.Dequeue()$
    $state(e) := Fetch$

SEND(d,value,e) $\equiv$
  if $\neg d.recvBuffer.isFull$ **then**
    $d.recvBuffer.Enqueue(value)$
    $state(e) := Fetch$

LogP

In 1993 Culler et al. proposed the LogP model [72], an asynchronous model of a distributed-memory multi-computer in which processors communicate by point-to-point messages. This model assumes that the parallel algorithms will need to be developed with a large amount of data elements per processor, and that the high latency and communication overhead, as well as limited bandwidth, will continue to be a problem. It is derived from BSP, resolving perceived problems with BSP's performance at router saturation. The main parameters of the model are: L (the latency of message passing); o (overhead of processor involved in message preparation and processing); g (the minimum time interval between successive messages, its inverse is essentially the bandwidth of the communication); P (the number of computers in this model). Thus, it is different from the BSP model with its additional parameter o, and different meaning of parameter L and g. L in LogP actually measures the latency of message passing, reflecting the nature of asynchronous execution. Usually with these four parameters, it is not easy to design algorithms on the LogP model.

Memory levels. LogP is a two-level memory model providing scope for strict locality.

Synchronization The model is asynchronous, i.e., processors work asynchronously and the latency experienced by any message is unpredictable, but it has L upper bound in the absence of stalls.

Communication It is assumed that the network has a finite capacity such that at most L/g messages can be in transit from any processor or to any processor at any time. If a processor attempts to transmit a message that would exceed this limit, it stalls until the message can be sent without exceeding the capacity limit. Because of variations in latency, the messages directed to a given target module may not arrive in the same order as they are sent. The basic module assumes that all messages are of small size. All processors are seen as being equally distant, so equally costly to communicate with.

Advantages. The LogP is architecture independent and it is essentially congruent.

Disadvantages. This model specifies the performance characteristics of the interconnection network, but does not describe the structure of the network. Indeed, it provides no scope for accounting for the lower communication costs between close processors. LogP does not abstract from parallelism, synchronization and communication due to its asynchrony and greater degree of flexibility.

ASM ground model. The LogP ASM model extends the PRAM one with features related to distributed memory. In this model each executor runs on a different RAM processor. An executor can access values residing in locations either in its local or remote memory units. In the former case, the rules performed are the same as those of the RAM model. In the latter case, executors exchange data values through message passing that is formalized with two new instructions: SEND and RECEIVE. The new set of valid instructions is:

$$\text{VALIDINSTR} = \langle \text{HALT}, \text{OTHER}, \text{LOAD}, \text{STORE}, \text{SEND}, \text{RECEIVE} \rangle$$

The data movement instructions must be changed in order to consider that different actions are required whether a data is in a local memory unit or in a remote one. This is formalized with the following function:

$$\textit{isLocal} : \text{UNITADDR} \times \text{EXECUTOR} \rightarrow \text{BOOLEAN}$$

that returns *true* if a given address is the local memory of a given executor. Therefore, the new rules for *LOAD* and *STORE* macros are:

LOAD(adr,reg,e) $\equiv$
  if $isLocal(adr, e)$ **then**
    $regfile(reg, e) := mem(adr)$
    $state(e) := Fetch$
  else
    $RECV(owner(adr), regfile(reg, e), e)$

STORE(adr,value,e) $\equiv$
  if $isLocal(adr, e)$ **then**
    $mem(adr) := value$
    $state(e) := Fetch$
  else
    $SEND(owner(adr), value, e)$

We abstract from communication details by the introduction of the message queue, called *recvBuffer*, where executors both store (enqueue) and load (dequeue) messages. The maximum length of *recvBuffer* is L/g as stated by this model definition. Before sending data values, an executor must check whether the destination can receive them or not, that is, *recvBuffer* is not full. Since we consider blocking communication instructions, if *recvBuffer* is full, that executor must wait (state(e) = *Execute*) until at least one position is free.
This is formalized with the following two macros:

RECV(s,destReg,e) $\equiv$
  if $\neg s.recvBuffer.isEmpty$ **then**
    $destReg := s.recvBuffer.Dequeue()$
    $state(e) := Fetch$

SEND(d,value,e) $\equiv$
  if $\neg d.recvBuffer.isFull$ **then**
    $d.recvBuffer.Enqueue(value)$
    $state(e) := Fetch$

1.4 Model of GPGPUs

In this section we define an ASM model of a real, mass-market architecture: the GPGPU. The GPGPU architecture does not present a new model of parallel execution. It is a special variant of the Private H-PRAM (recall Section 1.3), since

GPGPUs' cores are partitioned in a set of sub-PRAM, called CTA on Nvidia GPUs, such that each sub-PRAM has its own private contiguous block of shared-memory disjoint from the other ones in other sub-PRAMs. However, all sub-PRAM are equal in number of RAM and can not be changed at run-time. Moreover, as for the BSP when a given program P (i.e. *Kernel*) ends its execution a global check (i.e. bulk synchronization) is performed to ensure all sub-PRAM running P have finished.

Following the same approach of previous sections we provide an overview of its computational and architecture model, highlighting fundamental aspects, such as memory and synchronization management, and the communication cost. Moreover, we present GPGPUs implementations provided by the two main vendors: Nvidia and AMD. After that, we explain its formalization using the ASM model. In our explanation we follow a top-down approach, from the GPGPU computational model to its architecture model.

1.4.1 The computational model

GPUs vendors have introduced the unified shader architecture to support programmable shaders that can be used to express computations other than 3D graphics. GPU vendors create a programming toolchain capable of scheduling computations similar to those used for 3D shading (data-stream computations) on GPGPUs cores. A GPGPU is a compute device capable of executing a very large number of threads in parallel (to not be confused with operating system threads). It operates as a co-processor to the main CPU, or host. More precisely, a portion of an application that drives the computation of each core is commonly known as a *shader* (in the traditional 3D terminology) or *Kernel* (a term to stress the will to go beyond 3D graphics).

GPGPUs from both Nvidia and AMD vendors provide a computational model that has one *master process* executing on the CPU and driving one or more devices (GPUs); thus a GPU is a sophisticated co-processor of the CPU. A device is a set of computational processors capable of running *Kernels*, that is connected to two memory sub-systems: the CPU RAM, called *remote* in this context, and one *local* to the GPU. The *master process* can read and write to both *local* and *remote* memories of any device, and it is capable of querying the status of the completion of these tasks (i.e. reads and writes).

The program inputs and outputs can be set up to reside either in local or remote memory. The parallel computation is invoked by setting up one or more outputs and specifying a *domain of execution* for this output. In the case of a device having multiple processors (such as a GPU device), a scheduler distributes the workload region to various SIMD processors on the device. It is possible to schedule different *Kernels* on a GPGPU at once, allowing different input streams to be processed by a single application. Each physical processor in the GPGPU can execute a group of threads together called *wavefront* or *warp*.

Branch diversion

The effectiveness of an SIMD pipeline is based on the assumption that all threads running the same *Kernel* program expose identical control-flow behavior. In other words, a warp executes one common instruction at a time (i.e. single instruction counter), so full efficiency is realized when all threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjointed code paths.

Nvidia

The batch of threads that executes a *Kernel* is organized as a grid of cooperative thread arrays, called Cooperative Thread Arrays (CTA), also called *thread block*, as shown in Figure 1.1. Each thread has a unique thread identifier within the CTA. Programs use a data parallel decomposition to partition inputs, work, and results across the threads of the CTA. Each CTA thread uses its thread identifier to determine its assigned role, assign specific input and output positions, compute addresses, and select work to perform. The thread identifier is a three-element vector *tid*, (with elements *tid.x*, *tid.y*, and *tid.z*) that specifies the threads position within a 1D, 2D, or 3D CTA. Each thread identifier component ranges from zero up to the number of thread ids in that CTA dimension. Each CTA has a 1D, 2D, or 3D shape specified by a three-element vector *ntid* (with elements *ntid.x, ntid.y*, and *ntid.z*). The vector *ntid* specifies the number of threads in each CTA dimension. Each CTA has a unique CTA identifier (*ctaid*) within a grid of CTAs. Each grid of CTAs has a 1D, 2D , or 3D shape specified by the parameter *nctaid*. Each grid also has a unique temporal grid identifier (*gridid*). Threads may read and use these values through predefined, read-only special registers *%tid, %ntid, %ctaid, %nctaid*, and *%gridid*.

Threads are an abstraction provided by the programming infrastructure. At a lower level, the graphics hardware schedules the array of threads into the pool of physical processors until the input data is eventually processed. It is possible to schedule different *Kernels* on a GPU at once, allowing different input streams to be processed by a single application.

Threads within a CTA execute in SIMT (single-instruction, multiple-thread) fashion in groups called *warps*. A *warp* is a maximal subset of threads from a single CTA, such that the threads execute the same instructions at the same time. Threads within a warp are sequentially numbered. The warp size is a machine-dependent constant. Typically, a warp has 32 threads. The number of warp in execution at the same time is dependent on the active register usage of a *Kernel*. However, optimization of register usage only yields performance gains through better memory

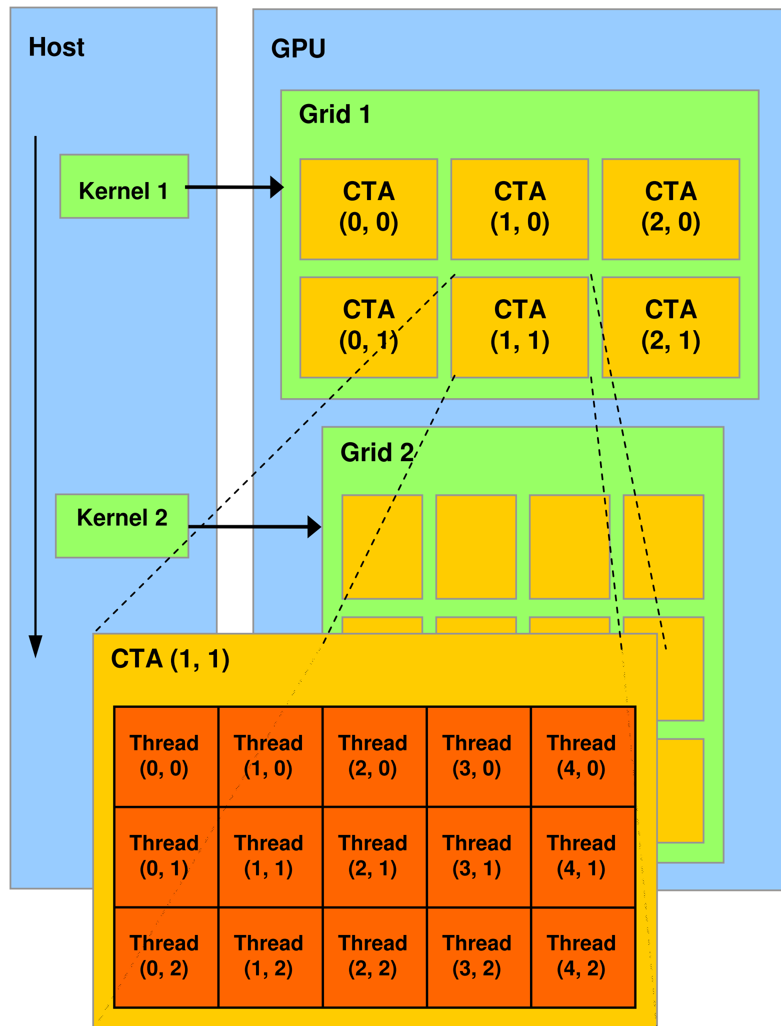

Figure 1.1: Nvidia GPU Computational model. A cooperative thread array (CTA) is a set of concurrent threads that execute the same kernel program. A grid is a set of CTAs that execute independently.

latency hiding. If the warp is not supported by a suitable number of active registers, the hardware will spill thread data into memory having a significant impact over performance.

Threads within a CTA can communicate with each other using a fast-shared memory. To coordinate the communication of the threads within the CTA, one can specify synchronization points where threads wait until all threads in the CTA have arrived. Threads in different CTAs can communicate and synchronize with each

other only, but only using atomic functions and accessing in global memory, that is 2 order of magnitude slower than shared memory, thus with an important impact over performance.

When a host program invokes a *Kernel* grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity, called Stream processors. The threads of a thread block execute concurrently on one Stream processor. As thread blocks terminate, new blocks are launched on the vacated Stream processors.

AMD

AMD GPGPU's computational model [20] is equivalent to Nvidia one, since arrays of input data elements stored in memory are mapped onto a number of SIMD engines, which execute *Kernels* to generate one or more outputs that are written back to output arrays in memory. The stream processor schedules the array of threads onto a group of threads processors, until all threads have been processed. Subsequent *Kernels* can then be executed, until application completes. Wavefronts are composed of quads, which are groups of 2x2 threads in the domain. Quads are processed together. If there are non-active threads within a quad, the processor that would have been mapped to those threads are idle. The main difference is related to the level of abstraction for *Kernel* threads organization. AMD hides every details of it, therefore programmers specify only the domain of execution, no other controls on execution configuration is provided.

1.4.2 The memory model

While the specific resources available in a given target GPU will vary, the kinds of resources will be common across platforms, and these resources are abstracted in the GPGPU memory model through state spaces and data types. A state space is a storage area with particular characteristics. All variables reside in some state space. A variable declaration describes both the variables type and its state space.

The characteristics of a state space include its size, addressability, access speed, access rights, and level of sharing between threads:

- `registers` are fast storage locations. The number of registers is limited, and will vary from platform to platform. When the limit is exceeded, register variables will be spilled to memory, causing changes in performance.

- The `global` state space is a memory that is accessible by all threads. For any thread all addresses in global memory are shared. It is not sequentially consistent, thus there can be race conditions between threads. Therefore, developers must adopt lock-free and wait-free style programming.

- The `local` state space is a private memory for each thread to keep its own data. It is typically standard memory with cache. The size is limited, as it must be allocated on a per-thread basis.

- The `shared` state space is a region of memory shared between a sub-set of threads. An address in shared memory can be read and written by any thread in the same sub-set.

Different implementations are provided by the two main GPUs vendors.

For instance, on Nvidia platforms, threads may access data from multiple memory spaces during their execution, as illustrated in Figure 1.2

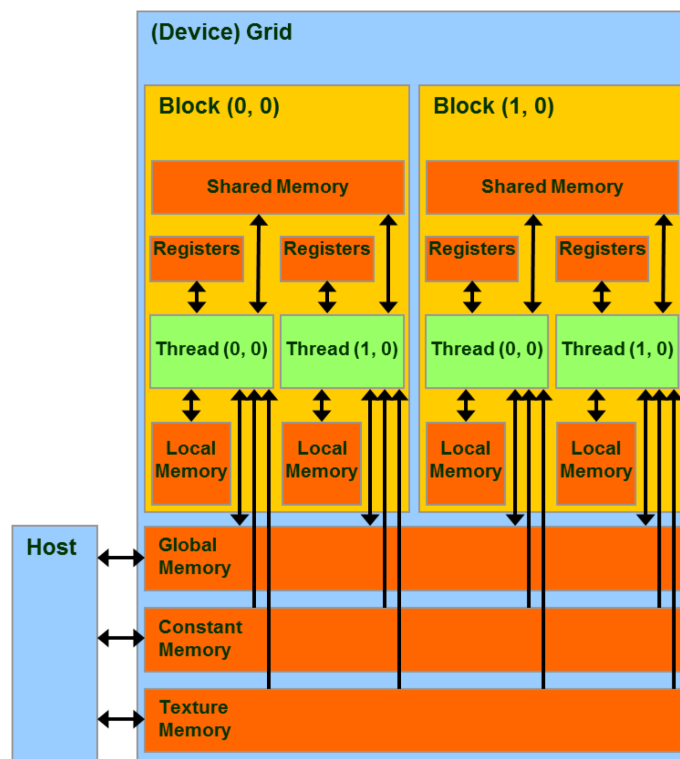

Figure 1.2: Nvidia Memory Hierarchy.

Each thread has a private local memory. Each thread block (CTA, see Figure 1.1) has a shared memory visible to all threads of the block and with the same lifetime as the block. Finally, all threads have access to the same global memory. There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages. Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats. The global, constant, and texture memory spaces are persistent across *Kernel* launches

by the same application. Both the host and the device maintain their own local memory, referred to as host memory and device memory, respectively, as shown in Figure 1.3.

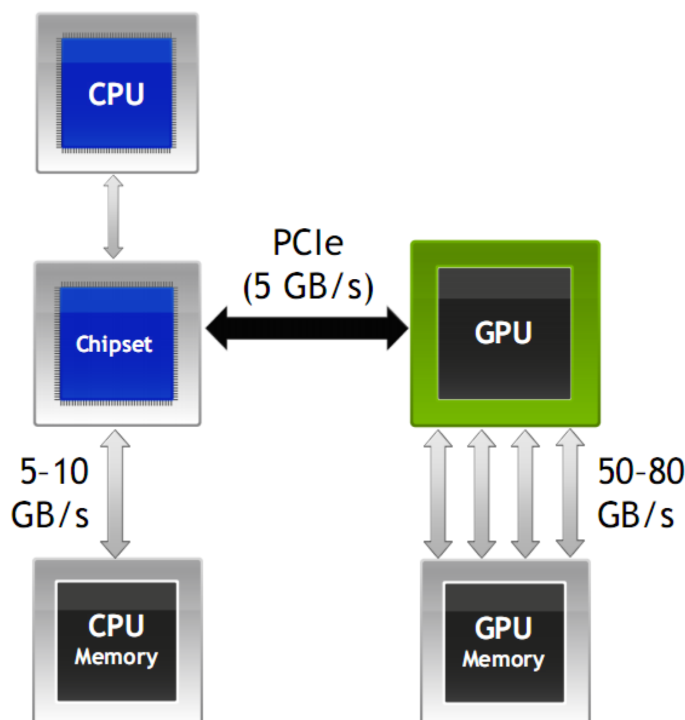

Figure 1.3: Architectural view of data transfer between CPU and GPU (with averaged observed bandwidth)

The device memory may be mapped and read or written by the host, or, for more efficient transfer, copied from the host memory through optimized API calls that utilize the devices high-performance Direct Memory Access (DMA) engine.

1.4.3 The architecture model

One of the main differences between GPUs and CPUs is that the former devote greater fraction of their transistor to arithmetic units, whereas CPUs devote them to cache. It is expected that this difference will continue in the future. This means that programmers will need to manage data locality much more carefully on future GPUs than they do on today's CPUs. Indeed, memory and communication models are the most important aspects of any parallel architecture, thus different processors have different design solutions. An architecture such as Cell BE has a non-cached, non-coherent[6] shared memory, so all data transfers between a core's

[6] It is not guaranteed the consistency of data stored in local caches of a shared memory.

small private memory and the global memory must be orchestrated through explicit memory-transfer commands. GPU architectures have two levels of non-coherent local memory (on-chip and off-chip). They cannot be accessed directly from the CPU, but only using dedicated hardware and executing dedicated instructions.
Therefore, three elements of an architecture model we are interested in: the structure of processor, the memory model and the CPU-GPU communication model.

AMD Stream Processor

The latest stream processor from AMD is called *Cypress*. As shown in Figure 1.4, Cypress has 20 SIMDs, each of which has 16 of what AMD calls Thread Processors (TP) inside it. Each of those TPs has 5 execution units, a branch prediction unit

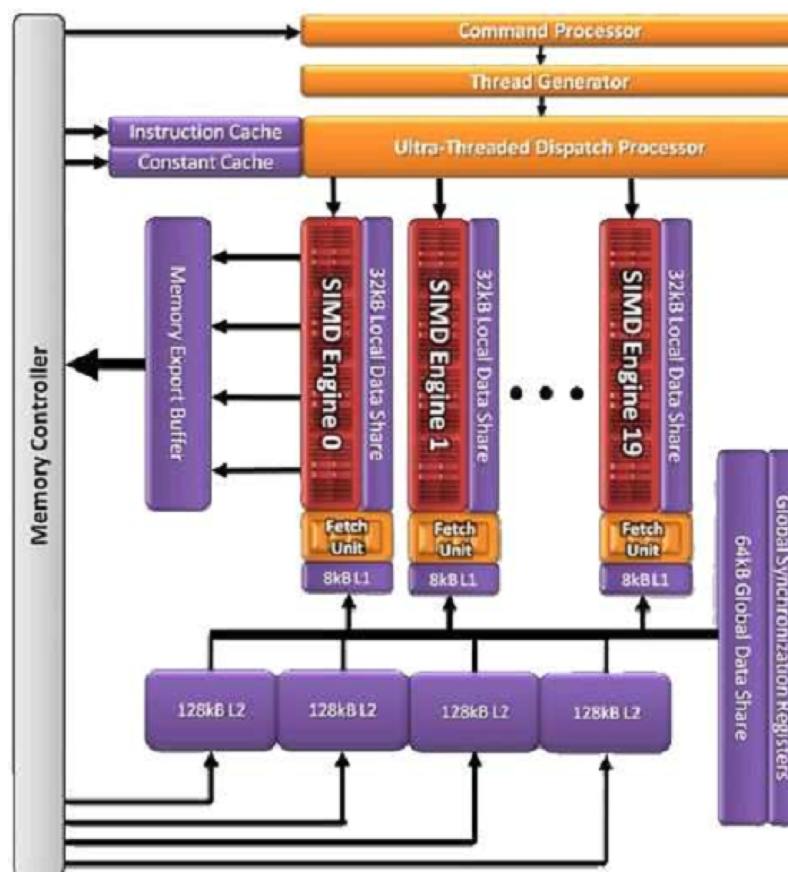

Figure 1.4: AMD Cypress architecture building blocks.

and a set of general purpose registers. Of the five units in each in each TP, there are four units which can handle a limited number of instructions per clock cycle and a fifth special function' unit. The four "standard" FPUs can handle the following instructions per clock cycle:

- Four 32-bit FP MAD
- Two 64-bit FP MUL or ADD
- One 64-bit MAD
- Four 24-bit Int MUL or ADD

The fifth “special function” unit is a superset of the other execution units in each TP but it can also handle more complex calculations such as integer division and bit shifting.

The command processor (CP) fetches command stream and data from dedicated registers mapped in CPU RAM, allowing the *master process* to drive the current device. The CP is DMA capable, so it can interact with host RAM without CPU arbitration. The Thread Generator (TG) is responsible for getting thread operations and data into the right formats for TPs (e.g. arranging data for optimal access), before passing down a thread batch for the dispatch hardware to execute. When TG completes, the Ultra-Threaded Dispatcher processor (UTDP) dispatches ready threads, which can be variable size in terms of objects, to SIMD engines for further processing. Each SIMD has a dedicated Fetch Unit that is responsible for requesting data from the memory controller and loading registers with returned data out of the cache.

On AMD platform the *wavefront* is a group of threads executed in lockstep on a single SIMD engine. A full *wavefront* contains up to 64 threads.

Memory. AMD equips each SIMD engine with one 8 KB L1 cache for storing texture and vertices, and one 32 KB local data store, called *Local Data Shared*, that is entirely under the programmer’s control. These two caches enable the TPs in the same SIMD engine to communicate to each other and share data without having to resort to off-die memory. For data sharing between threads executed on TPs of separate SIMDs, there’s the *Global Data Share*, a 64 KB memory that is not exposed by the AMD Stream SDK [20] yet, thus it’s up to the compiler to figure out how to best use it. In order to manage the inter-SIMD interaction, AMD Cypress includes Global Synchronization Registers, which means that programmers will can implement a synchronization primitive like a semaphore on a global level in the future.

Moreover, there is a second level in the cache hierarchy, a shared 512 KB (aggregate) L2 cache divided into 4 modules of 128 KB each which are accessed via a 1024-bit crossbar sitting between themselves and the L1s.

In an effort to satisfy requirements with regards to memory transfer correctness specific to GPGPUs, AMD Cypress supports the Cyclic Redundancy Checking (CRC) [73] to memory transfers.

The CPU processor units do not directly access GPU local memory; instead they issue memory requests through dedicated hardware units. There are two ways to

access memory - cached and uncached. Aside from caching, the main difference between the two is that uncached memory supports writes to arbitrary locations (scatter), whereas cached memory writes allow outputs to the associated domain location of the thread only.

Finally, AMD Cypress supports up to a total of 4 GB of GDDR5 DRAM memory on board.

CPU-GPU communication. All communication and data transfers between the system and GPU happen over the PCI-Express (PCI-e) channel, as shown in Figure 1.3. Transfers from the system to the GPU occur through the DMA engine. This DMA unit can run asynchronously from the rest of the GPU allowing parallel data transfers when the GPU array is busy running a previous GPU *Kernel*. Applications can request a DMA transfer from AMD CAL using special routines which allow copying data buffers between remote and device local resources.

nVidia GF100

The GF100 or Fermi based GPU is a MIMD array of SIMD processors, partitioned into Graphics Processing Clusters (GPCs), as illustrated in Figure 1.5. GPCs execute worked assigned by the GigaThread Engine scheduler, that is also responsible for creating and dispatching thread blocks in parallel. This scheduler is one of the most important technologies of the GF100 architecture because it allows to execute multiple *Kernels* of the same application on the GPU at the same time. However, *Kernels* from different application contexts still run sequentially.

Each GPC is armed with its own Raster Engine interfacing with up to four Stream Multiprocessors (SMs).

A SM comprises 32 CUDA cores (see in Figure 1.6). As described in [74], a CUDA core executes a floating point or integer instruction per clock through a fully pipelined integer arithmetic logic unit (ALU that supports full 32-bit precision for all instructions) and floating point unit (FPU that supports the IEEE 754-2008 floating-point standard). Therefore, GF100 provides the fused multiply-add (FMA) instruction for both single and double precision arithmetic.

Moreover, each SM has 16 load/store units (LD/ST), allowing source and destination addresses to be calculated for sixteen threads per clock. Supporting units load and store the data at each address to cache or DRAM. Special Function Units (SFUs) execute transcendental instructions such as sin, cosine, reciprocal, and square root. The SFU pipeline is decoupled from the dispatch unit, allowing the dispatch unit to issue to other execution units while the SFU is occupied.

Each SM features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently. GF100's dual warp scheduler selects two warps (of 32 threads each), and issues one instruction from each warp to a group of sixteen CUDA cores, sixteen load/store units, or four SFUs. Because warps

Figure 1.5: Block diagram of the Nvidia GF100 GPU

execute independently, GF100's scheduler does not need to check for dependencies from within the instruction stream.

Nvidia arms also each SM with a PolyMorph engine. This refers to a five stage piece of fixed-function logic that works in conjunction with the rest of the SM to fetch vertices, tessellate, perform viewport transformation, attribute setup, and output to memory. In between each stage, the SM handles vertex/hull shading and domain/geometry shading. From each PolyMorph engine, primitives are sent to the Raster Engine, each capable of eight pixels per clock (totaling 32 pixels per clock across the chip).

The general specifications and features of a compute device depend on its compute capability [19]. Devices with the same major revision number are of the same core architecture. The major revision number of devices based on the Fermi architecture is 2. Prior devices are all of compute capability 1.x.

Memory. Each of the 16 SMs has its own 64KB shared memory/L1 cache pool, which can either be configured as 16 KB shared memory/48 KB L1 or vice versa. This shared memory keeps data as local as possible, facilitating extensive reuse of

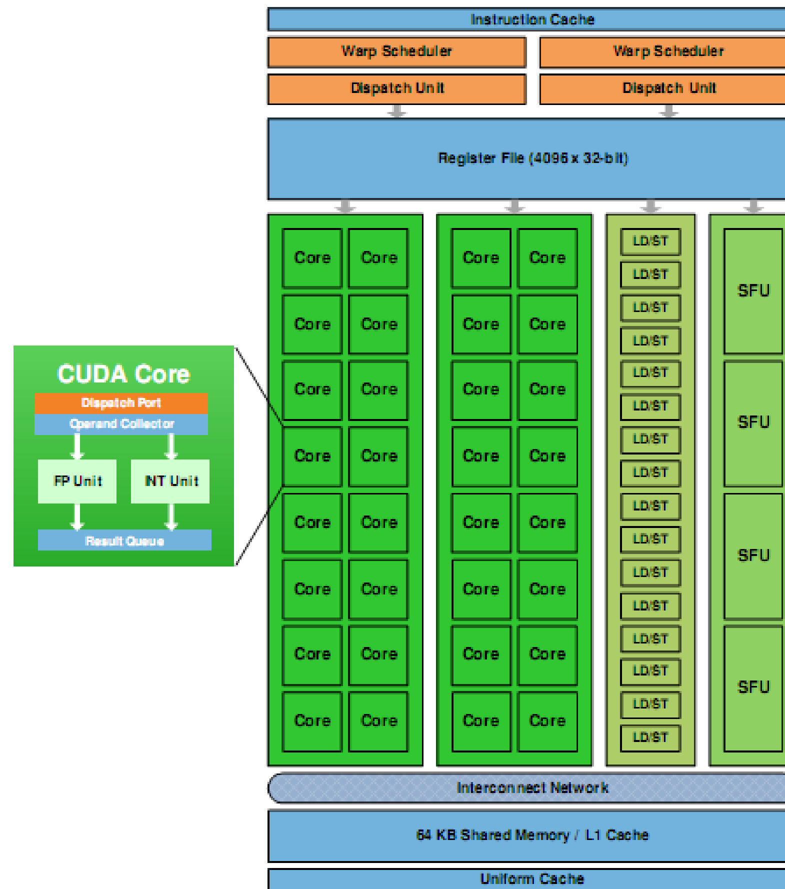

Figure 1.6: Nvidia Streaming Multi-processor architecture.

on-chip data, and greatly reduces off-chip traffic. Moreover, GF100 features a 768 KB unified L2 cache that services all load, store, and texture requests. This cache is read/write and fully coherent compared to previous Nvidia GPUs' versions where L2 cache was read-only. Finally, Nvidia GF100 supports up to a total of 6 GB of GDDR5 DRAM memory on board.

GF100 provides the Single-Error Correct Double-Error Detect (SECDED) ECC [75] based protection of data in memory. This corrects any single bit error in hardware as the data is accessed. In addition, SECDED ECC ensures that all double bit errors and many multi-bit errors are also be detected and reported so that the program can be re-run rather than being allowed to continue executing with bad data. Fermis register files, shared memories, L1 caches, L2 cache, and DRAM memory are ECC protected.

CPU-GPU communication. A host interface connects the GPU to the CPU via PCI-Express. Transfers management is the same of AMD GPUs.

1.4.4 ASM ground model

In this section we define a new ASM ground model for GPGPUs, based on ASM models presented in Section 1.3 and on GPGPUs memory model features. The memory access rule is the same of PRAM *CRCW*. For instance, on Nvidia GPUs, this is stated in Appendix G "Compute Capabilities" (CC) of the CUDA Programming guide [19]. For CC 1.x devices the specification states "if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing throughput by a factor equal to the number of separate memory requests. If the number of separate memory requests is n, the initial memory request is said to cause n-way bank conflicts. If a non-atomic instruction executed by a warp writes to the same location in shared memory for more than one of the threads of the warp, only one thread per half-warp performs a write and which thread performs the final write is undefined.". Whereas, having CC 2.0 devices, the specification states "A bank conflict only occurs if two or more threads access any bytes within different 32-bit words belonging to the same bank. If two or more threads access any bytes within the same 32-bit word, there is no bank conflict between these threads: For read accesses, the word is broadcast to the requesting threads; for write accesses, each byte is written by only one of the threads (which thread performs the write is undefined). "

In the GPGPU ASM model we define the following universes:

- UNITADDR is a memory unit.
- VAL is the data type held in either memory or register.
- REGISTER is the finite set of registers in the `RF` .
- OPCODE = {HALT, OTHER, LOAD, STORE}.
- INSTRUCTION is the instructions set of a processor.
- VALIDINSTR = {HALT, OTHER, LOAD, STORE}.
- EXECUTOR ⊆ AGENT, agent that execute a list of instructions selected from INSTRUCTION set.

and we introduce the following functions:

- static
  - *op* : INSTRUCTION → OPCODE returns the opcode of a given instruction.
  - *memUnitAdr* : INSTRUCTION → UNITADDR returns the address of the given instruction.

- *value* : INSTRUCTION → VAL returns the value of a given instruction's operand.
- *isGlobal* : UNITADDR → BOOLEAN returns whether a given address is in global memory or not (i.e. shared memory).

- controlled
  - *regfile* : REGISTER → VAL returns the value held in the given register
  - *mem* : UNITADDR → VAL returns the value held at the given address

- monitored
  - *legalPartition* : EXECUTOR × UNITADDR → BOOLEAN

$$legalPartition(e, adr) = \begin{cases} \mathit{true} & \text{if } e \text{ and } adr \text{ are relative to the same sub-PRAM} \\ \mathit{false} & \text{otherwise} \end{cases}$$

  - *legal* : INSTRUCTION × EXECUTOR → BOOLEAN

$$legal(instruction, e) = \begin{cases} \mathit{true} & \text{if } instruction \in \text{VALIDINSTR and} \\ & \quad legalPartition(e, memUnitAdr(instruction)) \\ \mathit{false} & \text{otherwise} \end{cases}$$

  - *canAccess* : UNITADDR × EXECUTOR → BOOLEAN
    that returns true if for some management of concurrency a given executor can access a given address. If the access to a given *adr* is not possible, what must be done is left undetermined in the model through the use of *choose* rule.

The main rule of `M` that provides a formal model of the GPGPU is the same of BSP model:

M(e) ≡
  case *state*(e) **of**
    Fetch : *FETCH*(e)
    Execute : *EXEC*(*currinstr*(e), e)
    Wait : *SYNC*(e)

where the macros are formalized with the following rules:
FETCH(e) ≡
  choose *instr* **in** INSTRUCTION **with** *legal*(*instr*, e) **in**
    currinstr(e) := *instr*
    state(e) := *Execute*

and

EXEC$(instr, e) \equiv$
case $op(instr)$ **of**
Halt : $HALT(e)$
Store : $STORE(memUnitAdr(instr), val(instr), e)$
Load : $LOAD(memUnitAdr(instr), reg(instr), e)$
Lock : $LOCK(memUnitAdr(instr), e)$
UnLock : $UNLOCK(memUnitAdr(instr), e)$
Send : $SEND(d, value, e)$
Receive : $RECV(s, destReg, e)$
Other : NOP

where

LOCK$(adr, e) \equiv$
NOP

UNLOCK$(adr, e) \equiv$
NOP

SEND$(d, value, e) \equiv$
NOP

RECV$(s, destReg, e) \equiv$
NOP

HALT$(e) \equiv$
$state(e) := Wait$

This global check must be performed whenever a processor ends its computation.

SYNC$(e) \equiv$
forall $executor$ **in** Executor **with** $executor \neq e$
if $state(executor) = Wait$ **then**
$state(e) := idle$

Only when all executors finish, their results can be transferred to the CPU.
As introduced in Section 1.4.3, GPGPUs have two levels of memory other than `RF`
. Which memory is accessed does not change how the `M`'s state is updated. For this reason we modify memory load and store macros as follows:

LOAD(adrS,adrD,e) $\equiv$
if $isGlobal(adrD)$ **then**
if $canAccess(adrS, e)$ **and** $canAccess(adrD, e)$ **then**
$mem(adrD) := mem(adrS)$
$state(e) := Fetch$
else
choose $s \in Fetch, Execute$ **in**
$state(e) := s$
else
$regfile(reg, e) := mem(adrS)$
$state(e) := Fetch$

STORE(adr,value,e) $\equiv$
if $canAccess(adr, e)$ **then**
$mem(adr) := value$
$state(e) := Fetch$
else
choose $s \in Fetch, Execute$ **in**
$state(e) := s$

The assumption for the initial state of `M` is:

- $state$(e) := $Fetch$
- $currinstr$(e) := $undef$

Chapter 2

Expressing concurrency paradigms on VEEs

Each model presented in the previous section tries to act as a bridging model between real architectures and parallel programming. As result, there are many different programming environments, each of which is specialized in a specific domain. In this thesis we do not formulate a bridging model but provide an approach to express each model on a VEE to overcome the diverging gap between the actual architectures and the abstract view offered by VEEs. The mapping between them must preserve the semantic and represents barriers implicitly using known VEE semantic elements already perceived by the programmers. Therefore, in this chapter we give an answer to the following questions:

1. Is it possible to expose underlying architecture features at the VEE level by using standard semantic objects of VEEs (e.g. static and instance fields, methods, etc.)? If yes, how?

2. Using those features exposed at VEE level, is it possible to express mapping between execution models introduced in Chapter 1 and a VEE, such as the CLI?

In the following sections, firstly we present a brief summary of VEEs' main design features and capabilities in order to make the thesis self-contained.

2.1 Virtual Execution Environment: design and capabilities

VEEs provide adaptable software suitable for today's rapidly changing, heterogeneous computing environment, since they makes aspects of the host execution environment virtual by interposing a layer that mediates program execution through

dynamically examining and translating a program's instructions before they are executed on the host CPU. The ideas of VEEs along with intermediate languages and language independent execution platforms have attracted researchers for a long time. Well known examples include `UCSD P-code` [76], `AS-400` [77], hardware emulators such as `VMWare` [78] and Xen [79], semi-platform independent programming language such as the Sun's JVM [32] and more recently Microsoft's Common Language Infrastructure (CLI) [1]. The main reasons why researchers are looking at alternative implementation paths for native compilers:

- Portability. By using an intermediate language `n` languages on `m` platforms can be implemented by `n + m` translator instead of `n * m` translators.
- Efficiency. By delaying the translation to a specific native platform as much as possible, the execution platform can make optimal use of the knowledge of the underlying architecture, or even adapt to the dynamic behavior of the program.
- Security. High-level intermediate code is more amenable to deployment and run-time enforcement of security and typing constraints than low level binaries.
- Interoperability. By sharing a common type system and high-level execution environment interoperability between different languages becomes easier than binary interoperability.
- Flexibility. Combining high level intermediate code with meta-data enables the construction of meta-programming concepts such as reflection, dynamic code generation, serialization, etc.

Both JVM and CLI virtual machines offer many services such as dynamic loading, garbage collection, Just-In-Time (JIT) compilation that allow to provide all features previously listed. These VMs are stack-based since their operations read values from a stack of operands and push the result on the stack. Many well-known languages adopt a stack-based VM: `F#` [80], OCaml [81], Python [82], and Muskel [41] are examples. An alternative to stack-based VMs are register-based VMs, these machines offer the abstraction of registers instead of a stack of operands to pass the values to the instructions; an example of register-based VM is the machine of Perl 6 called Parrot [83], Nvidia PTX [84], and AMD Intermediate Language [85]. Although both stack-based and register-based machines are Turing equivalent there always is a fervent debate among the implementers of which model offer better performance.

In this thesis we focus our attention on multi-threaded strongly-typed stack-based VMs, such as the JVM and the CLI. In particular we are interested on the latter's implementations, i.e. Microsoft CLR [33] and `Mono` [86], because it provides a standard specification [1] for executable code and the execution environment in which it runs. These VEEs hold information about the program types, and their

structures. In particular they are be able to reflect types and their methods to running programs. Using these information is possible by a meta-program to map differ concurrency models, as we will explain later. For these reason the CLR (and the JVM as well) is a good example of a Strongly Typed Execution Environment (STEE).

2.2 Strongly Typed Execution Environment

The STEE provides direct support for a set of built-in data types, defines a hypothetical machine with an associated machine model and state, a set of control flow constructs, and an exception handling model. Its execution model is type driven and guarantees that type of values can be always established and values are always accessed only using the operators defined on them.

2.2.1 Machine model and state

The STEE manages multiple concurrent threads of control (not necessarily the same as the threads provided by a host operating system) [1]. A thread can be viewed as a singly linked list of activation records, where an activation record is created and linked back to the current record by a method call instruction, and removed when the method call completes (either by a normal return, a tail call, or by an exception). It is usual, but not necessary, that the activation records of a single thread are allocated on a run-time stack. However, since the management of activation records is abstracted away in the CLI, and to avoid confusion, we shall use the term "stack" here exclusively to refer to the *evaluation stack* of the virtual machine. In this thesis we assume this level of abstraction of the execution system as machine model, since it hides loads of details such as registers, stack and heap implementation. Figure 2.1 illustrates the machine (global) state model, which includes threads of control, method states, and multiple heaps in a shared address space.

Method state and the evaluation stack

Method state describes the environment within which a method executes. Associated with each method state there are: an evaluation stack (EvalS), an array of local variables, and an array of arguments.

The EvalS contains intermediate values of the computation performed by the executing method (the operand stack in JVM terminology). The EvalS is made up of slots that can hold any data type. Most CLI instructions retrieve their arguments from the EvalS and place their return values on the stack. Arguments to other methods and their return values are also placed on the EvalS. The ECMA specification [1] (partition I) states some constraints on the EvalS:

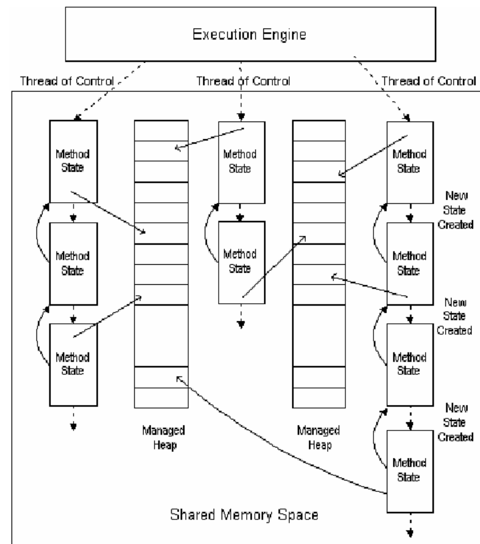

Figure 2.1: CLI Machine State Model

1. Depth and types of each element on the EvalS, at any given point in a program shall be identical for all possible control flow paths.

2. For every instruction the specification defines how values are pushed onto a stack or popped off a stack.

3. Backward branch constraints. If that single-pass analysis arrives at an instruction, call it location X, that immediately follows an unconditional branch, and where X is not the target of an earlier branch instruction, then the state of the evaluation stack at X, clearly, cannot be derived from existing information. In this case, the CLI demands that the evaluation stack at X be empty.

and guarantees the so-called *maxstack* property on the EvalS:

- Every method specifies a maximum number of items, called *maxstack*, that can be pushed onto the CIL evaluation stack. By analyzing the CIL stream for any method, it is easy to determine that number. However, specifying that maximum number ahead of time helps a CIL-to-native-code compiler in allocating internal data structures that model the stack and/or verification algorithm.

The local variables and arguments arrays, like the EvalS, can hold any single data type or an instance of a value type.

Moreover, associated with each method is meta-data that specifies:

- whether the local variables and memory pool memory will be initialized when the method is entered;

- the type of each argument and the length of the argument array;
- the type of each local variable and the length of the local variable array.

2.2.2 CLR compilation toolchain

Though other languages' code can be converted to run under JVM, they don't acquire true cross-language capabilities. Instead, the CLR is language neutral and enables multi-language applications to be developed. To assist this interoperability, a Common Type System (CTS) [1] is defined together with a Common Language Specification (CLS) [1] a set of rules that each component of the application must obey. Finally, the language used to write the instructions is called the Common Intermediate Language (CIL) [1]. Regardless of which high-level language you use, the result of compilation is a managed module. A managed module is a standard Windows portable executable (PE) file that requires the CLR to execute. The CLR does not actually work with modules; it works with assemblies. An assembly is a logical grouping of one or more managed modules or resource files. It represents the smallest unit of reuse, security, and versioning. In JVM terms an assembly could roughly be compared to a JAR file. In addition to emitting CIL, every compiler targeting the CLR is required to emit full meta-data into every managed module.

CIL code is sometimes referred to as managed code because the CLR manages its lifetime and execution. The CLR management includes, but is not limited to, three major activities: type control, structured exception handling, and garbage collection. A STEE can interpret the CIL or compile it using a Just-in-Time Compiler (JIT) into the native code of the underlying architecture. JVM is an interpreter that could rely on a JIT compiler to improve execution speed; CLR assumes that the CIL is always compiled before execution. The tool chain required to run code is illustrated in Figure 2.2.

The JIT compilation is done on demand, meaning that a method is compiled only when it is called. The compiled methods stay cached in memory, but they can be discarded if not used. If a method is called again after being discarded, it is recompiled.

2.2.3 Just In Time compilation

STEE compiles the CIL code into native CPU code using the JIT. CLR (as well as JVM) adopts a *two-stage* compilation. At the *1st* stage, a compiler that targets the CLR forms program files (i.e. assembly) in a standard machine independent format containing both code (i.e. intermediate language (CIL)) and meta-data. At the *2nd* stage, the JIT compiler converts the CIL as needed during execution and stores the resulting native code for subsequent calls. The loader creates and attaches a stub to each one of a type's methods when that type is loaded. On the initial

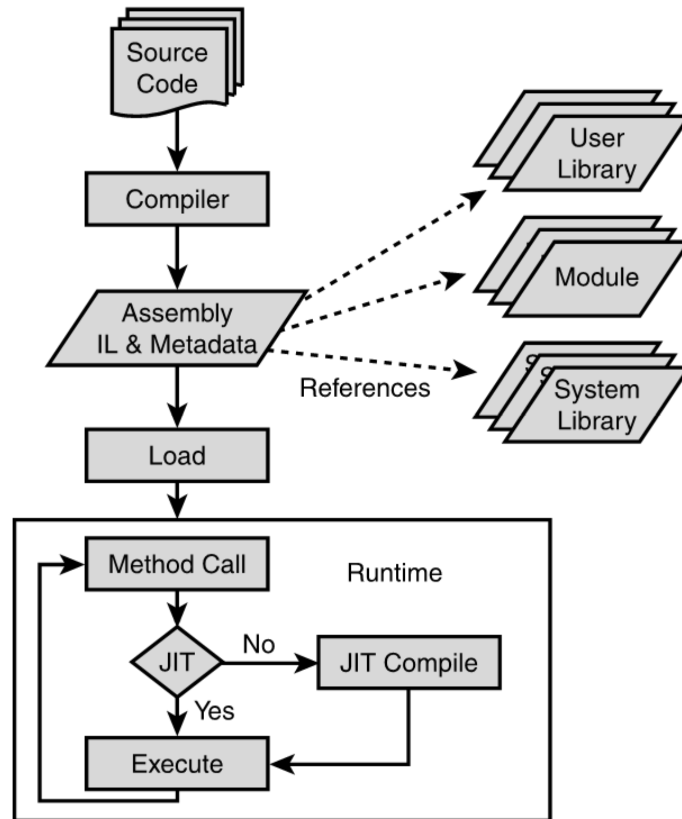

Figure 2.2: Basic CLR toolchain: from source code to its execution.

call to the method, the stub passes control to the JIT compiler, which converts the CIL for that method into native code and substitutes the stub with the native code location address. The choice of representing types and code in intermediate language form, rather than machine code, is somewhat constrained because of design goals. Without information on types it is almost impossible to have general support for dynamic loading of modules, thus reducing reuse of software. Furthermore, the compiler verifies the CIL code and relative meta-data it receives as input to find out whether the code is type safe or not, which means that it only accesses the memory locations it is authorized to access. A key aspect of the CLR programming model is the heavy reliance on meta-data.

2.2.4 Meta-data

Meta-data contains all the information necessary to describe and reference types defined by the type system. The concept of program meta-data arises naturally in those languages where programs are data, e.g. LISP. In these languages, the normal ways to describe relationships about different pieces of data can be used equally well to annotate programs with meta-data. However, program meta-data are common

in more traditional languages as well, although mostly in a limited way. Recently, the notion of program meta-data was widely used both in the design of languages (e.g. C#, Java) and in the corresponding execution environments (.NET CLR and JVM). These forms have important features:

- they are general purpose, i.e. the schema for their content can be defined by the programmer, and the mechanism to set and retrieve the content is not specific to a particular schema;
- they can be applied to generic program elements, with the programmer being able to declare specific restrictions about which class of elements can be designated as targets;
- they have customizable lifetime and location, encompassing all the range from source-only meta-data (as comments) to run-time meta-data (as typing information with reflection).

On VEEs, such as CLR and JVM, meta-data provides a common interchange mechanism for use between tools that manipulate programs (such as compilers, debuggers, and run-time code generators (RCG), as well as between those tools and the VEE. When types become a shared abstract ion between the execution environment and the programming language a larger amount of information is made available about a program to the run-time and to all the other programs interested in code analysis. Partial evaluators and programs can even execute these binaries with different semantics from the one of the execution environment. The simplicity of manipulate intermediate language and meta-data makes possible code analysis that would be hard to do in other contexts. Besides ordinary code analysis, the fact that types are shared between execution environment and the programming language implies that it is possible to provide libraries without implementation. The programmer makes use of such libraries and its invocation to library methods and types are used as placeholders into the binary format for further processing. After compilation programs may manipulate the output looking for special patterns inside the intermediate language, types and meta-data. The post processing may be done for several reasons: in [87] it is done for run-time code generation; a post processor would optimize patterns deriving from the use of domain specific operators; in [88] a meta-program processes sequential programs in their binary form and generates optimized parallel code; the executable is translated into an executable for a different platform.

As stated in [1], the CLI uses the meta-data to create instances of the types as needed and to provide data type information to other parts of the infrastructure (such as remoting services, assembly downloading, and security). After compilation, programs may manipulate the output looking for special patterns inside the CIL, types and meta-data. The .NET Framework provides an easy-to-use serialization mechanism for object implementers.

A key aspect of the CLR model's reflection is its extensibility. Indeed, it supports arbitrary meta-data attributes without introducing new keywords into the programming language. Together with the meta-data, required by the CLR for loading and managing the types contained within an binary file, it is possible to store arbitrary information in the form of `Custom Attributes` (`CA`). A `CA` is an instance of a class that inherits from the `Attribute` class. A `CA` is created by invoking one of its constructors, though all the values used to create it must be computable at compile time. `CA`s are serialized into an assembly at compile-time and ignored by the execution system. Nevertheless, the reflection `API` provides a means to retrieve these attributes at run-time. Follows a custom attribute written in C# that stores some tracking information about code modifications that you would typically record as comments in source code:

```
[AttributeUsage( AttributeTargets.Class | AttributeTargets.Method,
                 AllowMultiple = true)]
public class InfoTrackAttribute : System.Attribute
{
  private int infoID;
  private DateTime modificationDate;
  private string developerID;
  private string fixComment;
  public InfoTrackAttribute( string nInfoID, string nModificationDate,
                             string nDeveloperID )
  {
     this.infoID = nInfoID ;
     this.modificationDate = System.DateTime.Parse( nModificationDate )
         ;
     this.developerID = nDeveloperID ; }

     public string InfoID { get { return infoID; } }
     public DateTime ModificationDate {
                    get { return modificationDate.ToShortDateString();
                        }}
     public string DeveloperID {
                    get { return developerID ; } }
     public string FixComment {
                    get { return fixComment; } set { fixComment = value
                        ;}}
  }
}
```

2.2.5 Common Intermediate Language

CIL is a standard `ECMA` nr.335 [1] since 2005. There are several implementation of it. The two most important are the Microsoft MSIL [89] and the Mono CIL [86]. Consider the C# code in listing 2.1.

```
public int Fibonacci(int iteration)
```

```
{
  int previous = -1;
  int result = 1;
  for (int i = 0; i <= iteration; ++i)
  {
      int sum = result + previous;
      previous = result;
      result = sum;
  }
  return result;
}
```

Listing 2.1: C# source code that computes Fibonacci numbers

When the compiler compiles the C# code in Listing 2.1, a signature for the function is generated that looks like Listing 2.2.

```
.method public hidebysig instance int32 Fibonacci(int32 iteration) cil
    managed {
     .maxstack 2
     .locals init (
          [0] int32 previous,
          [1] int32 result,
          [2] int32 i,
          [3] int32 sum )
     {
```

Listing 2.2: CIL resulting code, method's signature.

Following the signature, the compiler computes slots that the execution engine requires. The amount of stack that is allocated for the method appears right after the signature declaration. Following the stack declaration, the compiler initializes (if needed) and declares each of the local variables that is required to evaluate this method. The resulting CIL looks something like the snippet of CIL code shown in Listing 2.2. Notice that this routine requires two stack slots to execute. One of the functions performed by the compiler is to determine how large the stack needs to be. For this example, it has been determined that no more than two stack slots will ever be required.

```
     L_0001: ldc.i4.m1
     L_0002: stloc.0
     L_0003: ldc.i4.1
     L_0004: stloc.1
     L_0005: ldc.i4.0
     L_0006: stloc.2
     L_0007: br.s L_0017
     L_0009: nop
     L_000a: ldloc.1
     L_000b: ldloc.0
     L_000c: add
     L_000d: stloc.3
```

```
    L_000e: ldloc.1
    L_000f: stloc.0
    L_0010: ldloc.3
    L_0011: stloc.1
    L_0012: nop
    L_0013: ldloc.2
    L_0014: ldc.i4.1
    L_0015: add
    L_0016: stloc.2
    L_0017: ldloc.2
    L_0018: ldarg.1
    L_0019: cgt
    L_001b: ldc.i4.0
    L_001c: ceq
    L_001e: brtrue.s L_0009
    L_0020: ret
```

Listing 2.3: CIL resulting code, method's body.

Arguments are one-based indexed and are referenced with `ldarg.`*x*. Every instance method has an "invisible" argument, called Argument Zero, not specified in the method signature: the class instance pointer, named *this*. Because static methods do not have such an "invisible" argument, for them argument number 0 is the first argument specified in the method signature. Local variables are zero-based and are referenced by `ldloc.`*x*. Listing 2.3 shows an example of the CIL code that is used to push an argument (argument 1) onto the stack and store it in a variable (location zero). It is important to note how the scope of variables is lost in the CIL code. It is not always possible to reconstruct it. A possible solution is proposed by Cisternino at al. in `a[C#]` [90]. `a[C#]` language extends the syntax of the C# language to allow programmers to annotate statements and code blocks using special custom attributes (e.g. Parallel), and retrieve them at run-time. Each of these custom attributes defines a scope inside the CIL code. However, a dedicated source-to-source compilation it is required to translate `a[C#]` code into standard C# code.

2.2.6 Delegate

They provide a mechanism for binding to a specific method on a specific target object. Informally, a delegate, i.e. an instance of a delegate type, is an object that points towards an invocation list of pairs of target objects and target methods. Upon the invocation of a delegate with a list of arguments, the methods in its invocation list are invoked sequentially with the corresponding target object and the given arguments, returning to the delegate caller the return value of the last method in the list.

Delegates are used in CLR-based libraries to represent the capability of calling a particular method. To that end, delegates are similar to a single-method interface,

the primary difference being that interfaces require the target method's type to have predeclared compatibility with the interface type. In contrast, delegates can be bound to methods on any type, provided that the method signature matches what is expected by the delegate type.

To instantiate a delegate, a method and optionally a target object reference are required. The latter is mandatory only when the binding is to an instance method. The *System.Delegate* type provides the *CreateDelegate* static method for creating new delegates that are bound to a particular method and object.

The following C# code uses the *CreateDelegate* method to bind a delegate to a static method and an instance method:

```
/* Delegate definition */
public delegate int BinaryOp(int x, int y);

/* Target class */
public class MathLib {
  internal int sum = 0;

  public int Add(int m, int n) {
    sum += m + n;
    return m + n;
  }

  public static int Subtract(int a, int b)
  { return a - b; }
}

class MyApp {
  static void Main() {
    MathLib target = new MathLib();

    Type tt = typeof(MathLib);
    Type dt = typeof(BinaryOp);

    BinaryOp op1 = (BinaryOp)Delegate.CreateDelegate(dt,
                                         tt, ``Subtract'');
    BinaryOp op2 = (BinaryOp)Delegate.CreateDelegate(dt,
                                         target, ``Add'');
  }
}
```

Calling *CreateDelegate* is an indirect way to invoke the delegate type's constructor. After the CLR has instantiated and bound a delegate to a method and object, the delegate's primary purpose is to support invocation. In C# delegate invocation resembles C-style function pointers:

```
static void Main() {
  BinaryOp op1 = new BinaryOp(MathLib.Subtract);
  int x = op1(3, 4);
}
```

2.2.7 Isolation and security boundaries

As for many programming technologies and environments, such as operating system (OS) with processes and the Java VM with the class loaders, the CLR defines its own unique model, called *Application Domain* (AppDomain), for providing isolation, security boundaries and resources ownership of an application from others applications. An AppDomain is a sub-process unit of isolation for managed code, which fills many of the same roles filled by an OS process since it provides a degree of fault and security isolation, and owns resources on behalf of the programs it executes. However, a process is an abstraction created by the OS, whereas an AppDomain is created by the CLR. This block of memory is then passed to the other AppDomain, which deserializes the block to produce a new object.

2.2.8 Communication inter-domain

Code in one AppDomain can communicate with types and objects contained in another AppDomain. However, the access to these types and objects is only through two well-defined mechanisms:

- By value. Types are marshaled by value across AppDomain boundaries. In other words, if an object is constructed in one AppDomain and a reference to this object is passed to another AppDomain, the CLR must first serialize the objects fields into a block of memory. This block of memory is then passed to the other AppDomain, which deserializes the block to produce a new object. The destination AppDomain uses the reference to this new object. The destination AppDomain has no access to the original AppDomains object.

- By Reference. Types that are derived from `System.MarshalByRefObject` can also be accessed across AppDomain boundaries. However, access to the object is accomplished by reference rather than by value. The .NET Remoting infrastructure employs the services of the `System.Runtime.Remoting.ObjRef` type.

2.2.9 Interoperability via the Platform Invocation Services

Managed code can easily call functions contained in DLLs using a mechanism called platform invocation or P/Invoke (for Platform Invoke). After all, many of the types defined in the .NET Class Library internally call functions exported from Kernel32.dll, User32.dll, and so on. Platform invoke relies on meta-data to locate exported functions and marshal their arguments at run time. It locates and invokes an exported function and marshals its arguments (integers, strings, arrays, structures, and so on) across the interoperation boundary as needed. The following illustration shows this process.

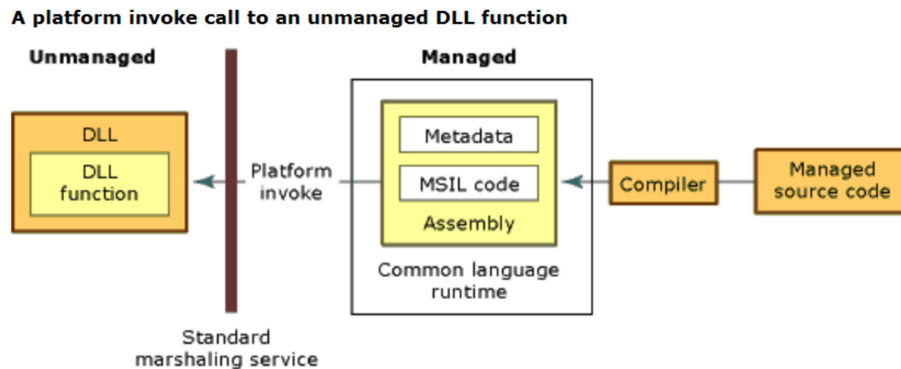

Figure 2.3: A platform invoke call to an unmanaged DLL function. Figure in the article "A Closer Look at Platform Invoke" of the .NET Framework Developer's Guide.

When platform invoke calls an unmanaged function, it performs the following sequence of actions:

1. Locates the DLL containing the function.
2. Loads the DLL into memory.
3. Locates the address of the function in memory and pushes its arguments onto the stack, marshaling data as required.
4. Transfers control to the unmanaged function.

Platform invoke throws exceptions generated by the unmanaged function to the managed caller. When a developer uses the CLR's P/Invoke mechanism to call a method, it is important to understand how it is possible to monitor and control the lifetime of an object; for instance what happen if the garbage collector decides to move/compact the object?

The CLR provides each AppDomain with a GC handle table. This table allows an application to monitor the lifetime of an object or manually control the lifetime of an object. When an AppDomain is created, the table is empty. Each entry on the table consists of a pointer to an object on the managed heap and a flag indicating how you want to monitor or control the object. An application adds and removes entries from the table via the `System.Runtime.InteropServices.GCHandle` type shown below. The GC handle table is used mostly in scenarios when developers are interoperating with unmanaged code. Basically, to control or monitor an object's lifetime, you call GCHandle's static Alloc method, passing a reference to the object that you want to monitor/control, and a GCHandleType, which is a flag indicating how you want to monitor/control the object. The GCHandleType is an enumerated type that defines two flags for monitoring (Weak and WeakTrackResurrection) and two for controlling (Normal and Pinned) lifetime of a object. For a complete description of the `System.Runtime.InteropServices` namespace see [91]. We are interested

in Pinned flag, since it allows you to control the lifetime of an object. Specifically, it informs the garbage collector that current object must remain in memory even though there might be no variables in the application that refer to this object. When a garbage collection runs, the memory for this object cannot be compacted (moved). This is typically useful when developer want to hand the address of the memory out to unmanaged code. The unmanaged code can write to this memory in the managed heap knowing that the location of the managed object will not be moved due to a garbage collection. This is used in **4-Centauri** to manage lifetime of objects that call Nvidia driver functions.

When a garbage collection occurs it marks all of the reachable objects. Then, the garbage collector scans the GC handle table; all Normal or Pinned objects are considered roots, and these objects are marked as well. Ate the end, the garbage collector compacts the memory, squeezing out the holes left by the unreachable objects, but pinned objects are not compacted (moved); the garbage collector will move other objects around them.

When a developer call GCHandle's static Alloc method, it scans the AppDomain's GC handle table, looking for an available entry where the address of the object he passed to Alloc is stored (e.g. Pinned), and a flag is set to whatever he passed for the GCHandleType argument. Then, Alloc returns a GCHandle instance back to him. A GCHandle is a lightweight value type that contains a single instance field in it, an IntPtr, that refers to the index of the entry in the table. When the unmanaged code calls back into managed code, the managed code would cast the passed IntPtr back to a GCHandle and then query the Target property to get the reference (or current address) of the managed object. When the unmanaged code no longer needs the reference, the GCHandle's Free method can be called, which will allow a future garbage collection to free the object (which also invalidates the instance by setting the IntPtr field to zero).

Actually when the CLR's P/Invoke mechanism is used to call a method, the CLR pins the arguments for you automatically and unpins them when the unmanaged method returns. However, the GCHandle type must be used explicitly when it is required to pass the address of a managed object to unmanaged code and then, the unmanaged function returns, but unmanaged code might still need to use the object later.

2.3 Meta-programming and Runtime code generation

In order to simplify access to special hardware through interoperability meta-programming techniques can be adopted; in particular the richness of program meta-data and the further ability to introduce custom annotations may drive the translation process as discussed in the rest of this dissertation.

Meta-programs are programs that manipulate other programs. Object-programs are programs manipulated by other programs. Program generators (PG) are meta-programs that produce object-programs as their final result. Writing programs that write code has numerous applications.

An example of application is when programs require to pre-generate tables of data for use at run-time. If a developer is writing a game and want a quick lookup table for the sine of all 8-bit integers, he can either calculate each sine himself and hand-code it, have his program build the table at startup at run-time, or write a program to build the custom code for the table before compile-time. While it may make sense to build the table at run-time for such a small set of numbers, other such tasks may cause program startup to be prohibitively slow. In such cases, writing a program to build static data tables is usually the best answer.

Another application is with large applications where many of the functions include a lot of boilerplate code. In this case a programmer can create a mini-language that will do the boilerplate code for him and allow him to code only the important parts. Now, if he can, it's best to abstract out the boilerplate portions into a function. But often the boilerplate code is not so pretty. Maybe there's a list of variables to be declared in every instance, maybe he needs to register error handlers, or maybe there are several pieces of the boilerplate that have to have code inserted in certain circumstances. All of these can make a simple function call impossible. In such cases, it is often a good idea to create a mini-language that allows developers to work with their boilerplate code in an easier fashion. This mini-language will then be converted into their regular source code language before compiling.

Another example of application is the programs execution optimization exploiting information available at runtime. The JIT compiler is an interesting example of that, since it compiles programs on the fly trying to produce efficient code in a small amount of time.

Modern bytecode execution environments with optimizing just-in-time compilers, such as JVM and CLR, provide an infrastructure for generating fast code at run-time. These platforms combine the advantages of bytecode generation (ease of code generation, portability) with native code generation (speed of the generated code). Moreover, both VMs incorporate a bytecode verifier that detects type errors and other flaws before executing the bytecode. While the resulting error messages are not particularly detailed they are more useful than a program crash that happens billions of instructions after the run-time was corrupted by flawed code generated at run-time.

Sestoft in [49] discusses how the JVM and the CLR make portable run-time code generation fairly easy and safe, and the generated code efficiently. This due mainly to the simplicity of generating stack-oriented portable bytecode, the support from bytecode verifiers, and highly optimizing just-in-time native code generators provided by both the JVM and the CLR.

Meta-programming is often used to overcome limitations of an existing programming language. Such limitations can either be performance or expressivity problems.

For instance, in [88] a meta-program, called Particular, transforms annotated programs relying on the reflection capabilities of the CLI infrastructure. Its transformation schema allows deferring the decision on how to render parallel a sequential program, considering the particular distributed/parallel architecture where it will be executed. In their approach, programmers can focus on functional aspects of the problem, relying on the well established programming toolchain for developing and debugging. Their assumption is that custom attributes will be enough to drive Particular in the parallelization of the program at a later stage.

2.4 Expressing architectural features

In literature, several approaches have been proposed to expose underlying architecture features at VEE level. In this section we survey the most relevant to our work.

2.4.1 Interface

In some domains interfaces are used to expose special domain features in an abstract way. For instance, in the distributed scenarios, distributed object technologies expose interfaces to allow objects running on a certain machine to be accessed from applications or objects running on other machines. Just as RPC makes remote procedures seem local, distributed object technologies make remote objects appear local. DCOM, CORBA, Java RMI, and .NET Remoting are examples of distributed object technologies, although they are implemented quite differently and are based on different business philosophies. Where possible, developers can factor out language artifacts specific to distributed programming and place them in a configuration layer.
The .NET Remoting [92] provides an abstract approach to interprocess communication that separates the remotable object from a specific client or server application domain and from a specific mechanism of communication. Remoting servers can also be any type of application domain. Any application can host remoting objects and provide its services to any client on its computer or network.

2.4.2 Custom annotation

The fundamental idea behind custom annotation consists of making data about the code available in the executable, thus at run-time. Custom annotations are interpreted by programs and are used for program transformation. Microsoft .NET provides support for implementing web services by means of custom attributes. A custom attribute named WebMethod is used to label methods that should be exposed as web services. A minimal web service written in C# that computes the sum of two integers is the following:

```
public class WSClass
{
  [WebMethod]
  public int add(int x, int y) { return x + y };
}
```

Once compiled, the *WSClass* type does not provide any web services interface. A different program, actually part of the Internet Information Server, is responsible for looking up reflection information within assemblies and generating a SOAP/WSDL interface to the method add over HTTP. A limit to the annotation model introduced by CLI is the granularity of annotations: they can only be used on methods and not inside to annotate code blocks. This is a pity because several programs whose goal consists of administering and manipulating other programs would benefit from a finer grain model for annotations. In [90] Cisternino et al. present an extension, called [a]C#, to the C# programming language supporting custom annotations on arbitrary code blocks or statements. The language extends the syntax of the C# language to allow a more general form of annotation and provides a run-time library that extends the reflection support with operations for retrieving the information about annotations inside methods. In particular they discuss the general operations on annotated code blocks used, which hide from the programmer the complexity of having to manipulate CIL instructions explicitly. Their approach required a source to source compiler, that reduces the extended model for custom annotations to the existing one with the help of some modification to the generated CIL. They encode information about ranges of code annotations by inserting placeholders into the bytecode. A placeholder is a dummy method that indicates the beginning and the end of an annotated block. Annotations are lifted onto methods and indexes are used to preserve the binding between dummy method calls and the relative annotations.

In [88] Cisternino et al. present an extension of the [a]C#, called *Particular*. They introduce two annotations, named *Parallel* and *Process*, that programmers use to provide hints to *Particular* on how to transform a sequential C# program into a parallel one. *Parallel* denotes the parts of the code subject to parallel execution, and the *Process* denotes the specific parts that have to be included in an independent control flow. Their approach is based on the manipulation of binary programs, therefore *Particular* can adopt strategies depending on the target architecture used for program execution. It may decide how annotations should be "better" rendered in the final program and what mechanisms should be used (threads, processes, etc.). The listing 2.4 shows that how to use these annotations in a sequential C# implementation of the Mandelbrot algorithm. Inserting a *Process* annotation inside the for loop entails a plane will be divided into stripes, such that each worker computes a different stripe.

```
public class MandelbrotFractal : IMandel{
  //...
  public void generateMandelbrot() {
```

```
    [Parallel(``Begin of a parallelizable block'')] {
      //initialize Mandelbrot parameters
      double gap = side / sizeW;
      double bb = ymax + gap;
      double xmin = x − side * 0.5;
      for( int wrk_id = 0; wrk_id < nr_stripes; wrk_id++) {
        [Process(``BeginofWorkerblock'')]{
          //Mandelbrot algorithm
          double aa = xmin − gap;
          bb −= gap;
          //...
        }
      }
      //update parameters for next plane
    }
    return;
  }
}
```

Listing 2.4: Mandelbrot set algorithm

The basic parallelism exploitation pattern used to handle *Parallel* annotated sections of code is the master/worker paradigm: a master process delivers “tasks” to be computed to a worker, picked up from a worker pool according to a proper scheduling strategy. The worker completely processes the task, computing a result which is eventually given back to the master process. This process is repeated until there are new tasks to be computed. In their approach each *Process* annotation leads to the generation of a master process/thread and of a set of worker processes/threads.

Their implement a meta-program that processes the sequential CIL code and for each annotation found it creates a method containing the CIL code of the annotated section (taking care of ensuring the appropriate handling of variables access).

```
public class MandelbrotFractal : IMandel
{
  //...
  public void generateMandelbrot() {
    // Parallel annotation
    Master0();
    return;
  }

  public void Master0() {
    // initialize Mandelbrot parameters
    double gap = side / sizeW;
    double bb = ymax + gap;
    double xmin = x − side * 0.5;
    AutoResetEvent[] Res = new AutoResetEvent(nr_stripes);

    for( int wrk_id = 0; wrk_id < nr_stripes; wrk_id++) {
      Res.Add(new AutoResetEvent(false));
      // ...
```

```
      // Process annotation
      Thread wrk = new Thread(new ThreadStart(Worker0));
      wrk.Start();
      // Update parameters for next plane
    }
    WaitHandle.WaitAll(Res);
  }

  public void Worker0() {
    double aa = xmin - gap;
    bb -= gap;
    // Mandelbrot algorithm
    //...
  }
}
```

Listing 2.5: Parallel version of the Mandelbrot set algorithm

The listing 2.5 shows C# code of parallel version of Mandelbrot algorithm that is equivalent to CIL code produced by *Particular*. The *Parallel* annotated block is replaced by an asynchronous method call to the correspondent (new) method, called *Master0*. Moreover, *Particular* emits the CIL instructions needed to call the (new) method (i.e. for reading actual parameters, to return values), called *Worker0*. Every new method code and meta-data are loaded into new library referenced by the original one.

2.4.3 Object system

Another approach to expose functional architectural features extends the object system with special types. For instance, Accelerator [37] is a library that uses data parallelism to program GPUs for general purpose uses under CLR. The main advantage is that no low-level aspect of the GPU is exposed to the programmers, only high level, data-parallel operations are: the programmer must use special data(-parallel) types to program the GPU instead of the CPU. Programmers don't need to learn graphics APIs nor convert their applications to use graphics pipeline operations, but can use stream programming abstractions of GPUs. The library implementation compiles the data-parallel operations on the fly into optimized GPU pixel shader code and API calls. All other operations are evaluated on the CPU. Following is an implementation[1] of a 2-dimensional convolution using Accelerator.

```
using Microsoft.Research.DataParallelArrays;

static float[,] Blur(float[,] array, float[] kernel) {
  float[,] result;
  DFPA parallelArray = new DFPA(array);
  FPA resultX = new FPA(0f, parallelArray.Shape);
```

[1] This code is Figure 2 in [37]

```
  for(int i = 0; i < kernel.Length; i++) {
    int[] shift Dir = new int[] {0,i};
    resultX += PA.Shift(parallelArray, shiftDir) * kernel[i];
  }
  FPA resultY = new FPA(0f, parallelArray.Shape);
  for(int i = 0; i < kernel.Length; i++) {
    int[] shiftDir = new int[] {i,0};
    resultY += PA.Shift(resultX, shiftDir) * kernel[i];
  }
  PA.ToArray(resultY, outresult);
  parallelArray.Dispose();
  return result;
}
```

The authors of Accelerator have developed a set of techniques to partition the operations into GPU pixel shader programs, such as combining element-wise operations, combining transformation operations, etc. They have demonstrated that:

- it is possible to compile high-level, data-parallel language extensions to mass-market parallel processors that are available today;
- it is worthwhile to explore compiling language extensions for data parallelism for GPUs.

Moreover, Microsoft has lately introduced parallel extensions, previously called `Parallel FX`, [93] to its .NET Framework technology in order to take advantage of the many-core hardware. They run on .NET 4.0, rely on features available in C# 4.0, and provide imperative data- and task-parallelism APIs in a declarative way. The .NET 4.0 introduces several new types:

- Data Structures for parallel programming. A set of concurrent collection classes, lightweight synchronization primitives, and types for lazy initialization. A programmer can use these types with any multi-threaded application code.
- Task Parallel Library (TPL) [94]. A library designed to write managed code that can automatically handle the partitioning of the work, the scheduling of threads on the ThreadPool, cancellation support, state management, and other low-level details. In listing 2.6, a C# implementation of the matrix multiplication is modified by replacing the outer for loop with static *Parallel.For* method, because the outer iterations are independent of one another.

```
using System.Concurrency;

void ParMatrixMult(int size, double[,] m1, double[,] m2, double[,]
    result)
{
  Parallel.For( 0, size, delegate(int i) {
    for (int j = 0; j < size; j++) {
```

```
        result[i, j] = 0;
        for (int k = 0; k < size; k++) {
          result[i, j] += m1[i, k] * m2[k, j];
        }
      }
    });
}
```

Listing 2.6: Parallel version of the Matrix multiplication algorithm

- PLINQ [95]. A library that provides a parallel implementation of LINQ to Objects. PLINQ implements the full set of LINQ standard query operators as extension methods for the System.Linq namespace and has additional operators for parallel operations. By default, PLINQ is conservative. At run time, the PLINQ infrastructure analyzes the overall structure of the query. If the query is likely to yield speedups by parallelization, PLINQ partitions the source sequence into tasks that can be run concurrently. If it is not safe to parallelize a query, PLINQ just runs the query sequentially. If PLINQ has a choice between a potentially expensive parallel algorithm or an inexpensive sequential algorithm, it chooses the sequential algorithm by default. When a programmer writes a query by invoking the *ParallelEnumerable.AsParallel* extension method on the data source, as shown in listing 2.7.

```
var source = Enumerable.Range(1, 10000);
var evenNums = from num in source.AsParallel()
               where Compute(num) > 0
               select num;
```

Listing 2.7: Parallel query

The *AsParallel* extension method binds the subsequent query operators, in this case, where and select, to the *System.Linq.ParallelEnumerable* implementations.

- DryadLINQ. It is a programming environment for writing large-scale data parallel applications running on large PC clusters. The goal of DryadLINQ is to make distributed computing on large compute cluster simple enough for general-application programmers too. DryadLINQ combines two Microsoft technologies: the Dryad distributed execution engine and the .NET Language Integrated Query (LINQ). The data model of DryadLINQ is typed .NET objects, and a DryadLINQ program is just a sequential program (written in C#, VB, or F#) composed of LINQ queries. For instance. the following listing is a complete implementation of the Map-Reduce computation framework in DryadLINQ

```
public static IQueryable<R>
  MapReduce<S,M,K,R>(this IQueryable<S> source,
```

```
                                    Expression<Func<S, IEnumerable<M
                                        >>> mapper,
                                    Expression<Func<M,K>> keySelector
                                        ,
                                    Expression<Func<K, IEnumerable<M>,
                                        R>> reducer)
{
    return source.SelectMany(mapper).GroupBy(keySelector, reducer)
        ;
}
```

2.4.4 Quotation

In `F#` meta-programming is supported by means of `F#` *quotations* (`<@ @>`). The quote operator instructs the compiler to generate data structures representing code rather than CIL. The F# parser and type checker statically guarantee the syntactic validity of quoted fragments and the typing of quoted literals. *Quotations* allow capturing of type-checked expressions as structured terms. They can be interpreted, analyzed and compiled to alternative languages. An interesting feature is that F# quotations can also be generated programmatically at run-time.

Quotations allow heterogeneous execution of F# programs since a single program written entirely in F# can run not only as .NET code, but also in various other environments. They make it possible to "take" part of the program, process it and execute it somewhere else. For example, in the following code a query comprehension is wrapped in `<@ @>` marks, which allows the SQL function to analyze the F# code and translate it to the appropriate SQL code.

```
let CustomersList =
  SQL <@ { for c in (db.Customers)
           when c.Country = "Italy"
           -> c } @>
```

This feature is very interesting for our aims since we want to process and analyze a high-level language implemented code and execute it where the best performance is provided; for example, matrix operations can be executed faster on a GPU rather than on a CPU.

2.5 Our approach: types + metadata

In the previous section emerges how STEE have a peculiar ability to treat code as data to some extent. The number of meta-programs inspecting and managing STEEs programs is steadily increasing relying on the presented techniques to acquire the information required to accomplish some task. STEE API exposes the Object-oriented design (OOD).

Primary Object-oriented programming (OOP) concepts such as objects, classes, inheritance, and dynamic typing were first introduced in the Simula [96] language and were initially intended to serve specific needs of real-world modeling and simulation. Object-orientation developed further as an independent general-purpose paradigm which strives to analyze, design and implement computer applications through modeling of real-world objects. Many real-world objects perform concurrently with other objects, often forming distributed systems. Because modeling of real-world objects is the backbone of the object-oriented paradigm and because real-world objects are often parallel, this paradigm needs to be extended with appropriate forms of parallelism. In fact, without that extension, when analysis and design models are taken into consideration, the notion of parallelism could be thrown away. In this case, what looked like a parallel solution is flattened into a single flow of control, as stated by the Von-Neumann model. Besides, this activity of flattening a concurrent model into a sequential application is far from trivial and is the major cause of programming errors.

Software objects are conceptually similar to real-world objects: they consist of state and related behavior. An object stores its state in fields and exposes its behavior through methods. Classes, and their fields and methods have access levels (e.g. static, private, public) to specify how they can be used by other objects during execution. In the same way each model of parallel computation specifies how different levels of memory can be used by executors, as explained in Section 1.2. Therefore, our idea is to map an access level to a level of memory, as shown in Figure 2.4, such that programmers can express at VM level the memory hierarchy of a model of parallel execution.

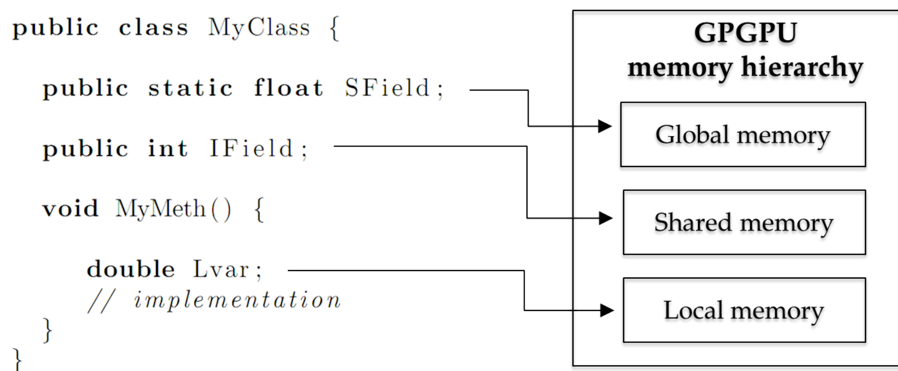

Figure 2.4: Mapping a C# class members to the GPGPU memory hierarchy.

To explain our solution, we think it is important to start from the description of the most relevant elements in a program. Later, we specialize them for exposing the underlying architecture features at STEE level.

2.5.1 Definitions

Let P be a program that executes on one of the architectures introduced in Section 1. The behavior of P is described by its code, that is a finite list of instructions according to the programming language used to implement P. We introduce $\mathbb{I}$ the domain of that programming language instructions, $\mathbb{IL}$ the domain of list of instructions and $\mathbb{T}$ the domain of the types recognized by the underlying architecture.

P will access a sub-set of registers and a sub-set of local memory units[2]. Moreover, it will share data with both other programs through either a shared memory or message passing communication. Therefore, let $\mathbb{RG}$ be the domain of the registers available in the underlying architecture `RF` , $\mathbb{LM}$ the domain of the local memory units referred to by an instruction or an instruction list, $\mathbb{GM}$ the domain of the global memory units referred to by an instruction or an instruction list. We define the domain of memory resources as union type:

$$\mathbb{MEM} = \mathbb{RG} \cup \mathbb{LM} \cup \mathbb{GM}$$

Moreover, we introduce $\mathbb{COM}$ the domain of communication units (e.g. input/output register).

The number programs that can be executed simultaneously depends on the underlying available processors. We introduce $\mathbb{PR}$ the domain of processors models:

$$\mathbb{PR} = \begin{cases} RAM \\ PRAM \\ H-PRAM \\ LogP \\ BSP \end{cases}$$

To summarize this description in a more concise way, we introduce the following notation for programs.

Definition 2.5.1. *A program P is a tuple $P \equiv \mathbb{IL} \times \mathbb{MEM} \times \mathbb{PR} \times \mathbb{COM}$.*

2.5.2 Definitions for STEEs

Considering now a program P_{STEE} written in one of the programming languages targeting a STEE. It exposes the Object-Oriented programming model, thus its behavior is described by a set of methods. Let $\mathbb{C}_{STEE}$ be the domain of the classes and $\mathbb{MET}_{STEE}$ the domain of the methods, we define the follows

$$met_{STEE} : \mathbb{C}_{STEE} \rightarrow 2^{\mathbb{MET}_{STEE}}$$

[2]A memory unit is either a page or a segment of memory. This depends on the specific architecture model implementation.

as the function that given a class $c \in \mathbb{C}_{STEE}$ returns a set of all the methods $m_i \in \mathbb{MET}_{STEE}$ declared in c, defined as

$$met_{STEE}(c) = \{m_1, m_2, ..., m_n\}$$

Then we define the following:

$$\eta_{STEE} : \mathbb{MET}_{STEE} \to \mathbb{IL}_{STEE}$$

as the function that given a method $m \in \mathbb{MET}_{STEE}$ returns its list of instructions.

Each object maintains its state in one or more fields, in methods' parameters and variables defined inside methods. These parts of an object state differ for their accessibility; while instance fields are shared between all instances of the same class, methods' parameters are shared only by the caller and receiver objects. In Object-Oriented programs, developers can use different scopes for variables to define how a community of objects can provide their services and how they can interact to perform actions used by other members of the community. We introduce $\mathbb{LOC}_{STEE}$ the domain of variable names defined in a method and referred to by an instruction or an instruction list, $\mathbb{MPRM}_{STEE}$ the domain of method's parameters, $\mathbb{FLD}_{STEE}$ the domain of all fields variable names in a type. Therefore, we define the following functions on these domains:

- $mlocals_{STEE} : \mathbb{MET}_{STEE} \to \mathbb{LOC}_{STEE}$
  as the function that given a method $m \in \mathbb{MET}_{STEE}$ returns its set of local variables;

- $params_{STEE} : \mathbb{MET}_{STEE} \to \mathbb{MPRM}_{STEE}$
  as the function that given a method $m \in \mathbb{MET}_{STEE}$ returns its set formal parameters;

- $fields_{STEE} : \mathbb{MET}_{STEE} \to \mathbb{FLD}_{STEE}$
  as the function that given a method $m \in \mathbb{MET}_{STEE}$ returns the set of fields referred by m;

Finally, the domain of all memory resources referred to by a method is

$$\mathbb{MEM}_{STEE} = \mathbb{LOC}_{STEE} \cup \mathbb{MPR}_{STEE} \cup \mathbb{FLD}_{STEE}$$

A P_{STEE} runs on a hypothetical machine with an associated model provided by a STEE, e.g. Von-Neumann model. At source level, programmers design their algorithms leveraging the multi-threaded architecture of a STEE. We introduce $\mathbb{PR}_{STEE}$ the domain of threads that will execute on the STEE. We assume the function that maps a $t \in \mathbb{PR}_{STEE}$ on real available processors is provided by the framework.

There are many frameworks (e.g. RMI on Java, .NET Remoting) that allow to execute remote calls and exchange data in a distributed system. Let $\mathbb{COMCS}_{STEE} \subset$

T be the domain of special classes, called remotable class, whose methods can be invoked by remote objects (recall Section 2.2.8).

To summarize, a program that executes on a STEE is a tuple:

$$P_{STEE} \equiv \mathbb{MET}_{STEE} \times \mathbb{MEM}_{STEE} \times \mathbb{PR}_{STEE} \times \mathbb{COMCS}_{STEE}$$

Considering the following example where the STEE is CLR and the programming language is C#. It provides a sequential implementation of the Mersenne-Twister [97, 98] pseudo-random number generator, that will be used as a benchmark in Section 5.2. The coefficients used in listing 2.8 are defined by Matsumoto and Nishimura in [98].

```
public class MersenneTwister
{
  /* Period parameters */
  private const int N = 624; // 624 length array to store the state of
     the generator
  private const int M = 397;

  /* the array for the state vector  */
  private readonly int[] _mt = new UInt32[N];
  private int _mti;

  private static readonly int[] _mag01 = { 0x0, 0x9908b0df };

  public int Generate(int seed)
  {
    int y;

    _mt[0] = seed & 0xffffffffU;

    /* Initialize current state */
    for (_mti = 1; _mti < N; _mti++) {
      _mt[_mti] = (int)(1812433253 * (_mt[_mti - 1] ^ (_mt[_mti - 1] >>
          30)) + _mti);
      _mt[_mti] &= 0xffffffffU;
    }

    /* Generate N words at one time */
    if (_mti >= N) {
      int kk = 0;
      for (; kk < N - M; ++kk) {
        y = (_mt[kk] & 0x80000000) | (_mt[kk + 1] & 0x7fffffff);
        _mt[kk] = _mt[kk + M] ^ (y >> 1) ^ _mag01[y & 0x1];
      }
      for (; kk < N - 1; ++kk) {
        y = (_mt[kk] & 0x80000000) | (_mt[kk + 1] & 0x7fffffff);
        _mt[kk] = _mt[kk + (M - N)] ^ (y >> 1) ^ _mag01[y & 0x1];
      }
```

```
            y = (_mt[N - 1] & 0x80000000) | (_mt[0] & 0x7fffffff);
            _mt[N - 1] = _mt[M - 1] ^ (y >> 1) ^ _mag01[y & 0x1];
            _mti = 0;
        }

        /* Tempering transformation */
        y = _mt[_mti++];
        y ^= y >> 11;
        y ^= (y << 7) & 0x9d2c5680;
        y ^= (y << 15) & 0xefc60000;
        y ^= y >> 18;

        return y;
    }
}
```

Listing 2.8: C# sequential implementation of the Mersenne-Twister pseudo-random number generator.

There is only one method, $Generate \in \mathbb{MET}_{CLR}$. It accesses $mlocals_{CLR}(Generate) = \{y, kk\}$ local variables, it has not parameters, $params_{CLR}(Generate) = \{seed\}$, it refers $fields_{CLR}(Generate) = \{N, M, _mt, _mti, _mag01\}$ fields.

Since this example implements a sequential version of the Mersenne-Twister algorithm, it will execute on a single RAM processor, that is $\mathbb{PR}_{CLR} = \{\text{RAM}\}$. No communication with other objects is implemented, thus $COMCS_{CLR} = \oslash$. Finally, there are no custom attributes. This description in our notation results:

$$MersenneTwister_{CLR} \equiv \mathbb{MET}_{CLR} \times \mathbb{MEM}_{CLR} \times \mathbb{PR}_{CLR} \times \mathbb{COMCS}_{CLR}$$

Before the execution, a P_{CLR} must be translated into an intermediate form $\mathtt{P}'_{CLR}$, that is CIL. Since the CLR types are a shared abstraction between the execution environment and the programming language, a larger amount of information is made available about a program to the run-time and to all the other programs interested in code analysis and manipulation, as shown in [90, 99].

In the following sections we will leverage these CLI features to define a set of types, meta-data, and functions that map elements of P'_{CLI} to elements of another program P''_{CLI} that will run on an architecture with a different model of parallel execution. The equivalence between P'_{CLI} and P''_{CLI} must be guaranteed by the compiler that implements those mapping functions.

It is important to highlight that in the following sections we map an executor defined in one ASM model (recall Sections 1.2 and 1.3) to a thread running on RAM processor.

In chapter 4 we will provide a complete and concrete example of expressing the GPGPU model of execution on CLI, along with a compiler **4-Centauri** that implements the mapping between those models.

2.5.3 Expressing the RAM on the CLI

We express each memory resource in RAM model, i.e. $\mathbb{MEM}_{RAM} = \mathbb{RG} \cup \mathbb{LM}$, on the CLI as a binding between variable names of class members and elements in $\mathbb{MEM}_{RAM}$.

Let $c \in \mathbb{T}$ be a class that we define for mapping purposes. The set of methods defined in c, and given by $met(c)$ will be the code executed on RAM. In the rest of this thesis, we assume that for each class introduced for mapping purposes corresponds a distinct AppDomain (recall Section 2.2.7). An AppDomain may have multiple threads objects. These threads execute methods of objects and refer objects that reside in exactly that AppDomain.

We introduce the following functions that bind variable names to memory units in $\mathbb{MEM}_{RAM}$.

$$\rho : \mathbb{LOC}_{CLI} \rightarrow \mathbb{MEM}_{RAM}$$

it binds method's local names to registers $\in \mathbb{RG}$ of $\mathbb{MEM}_{RAM}$. The idea is that RAM registers can be accessed only by the executor currently running on that processor as well as a method's local variable is accessed only by the thread executing that method.

$$\omega : \mathbb{MPRM}_{CLI} \rightarrow \mathbb{MEM}_{RAM}$$

it binds method's formal parameter names to registers $\in \mathbb{RG}$ of $\mathbb{MEM}_{RAM}$. As for local variables, method's formal parameters variables are accessed only by the thread executing that method.

$$\lambda : \mathbb{FLD}_{CLI} \rightarrow \mathbb{MEM}_{RAM}$$

it binds field names to local memory unit $\in \mathbb{LM}_{RAM}$ of $\mathbb{MEM}_{RAM}$. These variables can be accessed only by a thread running an instance of c.
Finally,

$$\mu_{RAM}(c) = \bigcup_{m \in met_{CLI}(c)} (\rho(mlocals_{CLI}(m)) \cup \omega(params_{CLI}(m)) \cup \lambda(fields_{CLI}(m)))$$

$$\mu_{RAM} : ID \rightarrow MEM_{RAM}$$

is the function that given a class $c \in T$, for each method $m \in met_{CLI}(c)$ maps all variable names referred in m to memory resources in the RAM memory model.

$\mu_{RAM}(c)$ is applied to every methods in $met_{CLI}(c)$. However, through annotations programmers can select a sub-set of them for mapping purposes. For this

reason we introduce a custom attribute, called *Kernel*.

Meta-data. Let $\mathbb{A}$ be the domain of meta-data and $\mathbb{AS}$ the domain of a meta-data set, we introduce the follows:

$$isKernelAnn_{CLI} : \mathbb{MET}_{CLI} \rightarrow \mathbb{B}$$

boolean function that given a method $m \in \mathbb{MET}_{CLI}$ is defined as

$$isKernelAnn_{CLI}(m) = \begin{cases} true & \text{if } m \text{ is annotated by } Kernel \text{ attribute} \\ false & \text{otherwise} \end{cases}$$

Therefore, $\mu_{RAM}(c)$ is modified as follows:

$$\mu'_{RAM}(c) = \bigcup_{\substack{m \in met_{CLI}(c) \wedge \\ isKernelAnn_{CLI}(m)}} \left(\begin{array}{l} \rho(mlocals_{CLI}(m)) \cup \omega(params_{CLI}(m)) \\ \cup \lambda(fields_{CLI}(m)) \end{array} \right)$$

2.5.4 Expressing the PRAM on the CLI

On the PRAM multiple executors act concurrently, one for each RAM processor. They access, modify and exchange data values through a global memory. This memory model can be mapped on the CLI as a single class instantiated by multiple threads in the same AppDomain.

Therefore, the domain of memory resources for the PRAM memory model extends the RAM with a global memory, such that $\mathbb{MEM}_{PRAM} = \mathbb{RG}_{PRAM} \cup \mathbb{LM}_{PRAM} \cup \mathbb{GM}_{PRAM}$. For this reason we must replace the domain of all fields $\mathbb{FLD}_{CLI}$ with $\mathbb{IFLD}_{CLI}$ the domain of instance field variable names in a type and $\mathbb{SFLD}$ the domain of static field variable names in a type. Therefore, we add the following functions on those new domains:

- $ifields_{CLI} : \mathbb{MET}_{CLI} \rightarrow \mathbb{IFLD}_{CLI}$
  given a method $m \in \mathbb{MET}_{CLI}$ returns the set of instance fields referred by m;

- $sfields_{CLI} : \mathbb{MET}_{CLI} \rightarrow \mathbb{SFLD}_{CLI}$
  given a method $m \in \mathbb{MET}_{CLI}$ returns the set of static fields referred by m.

and a binding function for each new domain:

$$\lambda : \mathbb{IFLD}_{CLI} \rightarrow \mathbb{MEM}_{PRAM}$$

it binds instance field names to local memory unit $\in \mathbb{LM}_{PRAM}$ of $\mathbb{MEM}_{PRAM}$. These variables can be accessed only by a thread executing methods of an instance of c.

$$\sigma : \mathbb{SFLD}_{CLI} \rightarrow \mathbb{MEM}_{PRAM}$$

it binds static field names to memory units $\in \mathbb{GM}_{PRAM}$ of $\mathbb{MEM}_{PRAM}$. The idea is that PRAM global memory units can be accessed by all executors running on PRAM as well as a class's static fields are accessed by all threads executing methods of a class.

The function that given a class $c \in T$, for each method $m \in met_{CLI}(c)$ maps all variable names referred in m to memory resources in the PRAM memory model is:

$$\mu_{PRAM}(c) = \mu_{RAM}(c) \cup (\bigcup_{m \in met_{CLI}(c)} \sigma(sfields_{CLI}(m)))$$

$$\mu_{PRAM} : ID \rightarrow MEM_{PRAM}$$

Meta-data. Considering the *Kernel* annotation introduced in the previous section, $\mu_{PRAM}(c)$ is modified as follows:

$$\mu'_{PRAM}(c) = \mu'_{RAM}(c) \cup (\bigcup_{\substack{m \in met_{CLI}(c) \wedge \\ isKernelAnn_{CLI}(m)}} \sigma(sfields_{CLI}(m)))$$

Example of implementation: data-parallel on PRAM

In order to give an idea of a possible implementation of formal concepts expressed in previous sections, we provide a concrete example: data-parallel operations implemented in C# but considering PRAM as execution model. In listing 2.9, we implement a method for each data-parallel operation: Map and Reduce. Denoting with *outputs* the list of resulting values, and given a list of input data, *inputs* = $\{x_1, x_2, \cdots, x_n\}$

- *Map* is the operation that applies a function to every element in *inputs*. Informally, we have

$$outputs = Map(f, [x_1, x_2, \cdots, x_n]) = [f(x_1), f(x_2), \cdots, f(x_n)]$$

- *Reduce* is the operation that collapses a set into a single value by repeated application of some associative binary operator.

$$output = Reduce(\otimes, [x_1, x_2, \cdots, x_n]) = x_1 \otimes x_2 \otimes \cdots \otimes x_n$$

```
/* Type of values to compute */
public class Data<T>
{
  T[] _data;
  int _index;

  public Data(int capacity) { ... }

  public Data(T[] array) {
    _index = 0;
    _data = array;
  }

  public T Current {
    set {
      if(_index < _data.Length)
         _data[_index] = value;
      else
         throw new ArgumentOutOfRangeException();
    }

    get {
      if (_index < _data.Length)
         return _data[_index];
      else
         throw new ArgumentOutOfRangeException();
    }
  }

  public bool MoveNext() {
    return ++_index < _data.Length;
  }

  public void Reset() {
    _index = 0;
  }

  /* Array length */
  public static int GetSize() {...}
}

/* Available operations for reduction purposes */
public enum DPOper {
  Add = 1,
  Prod = 2
}

/* Example of delegate definition */
public delegate void DPD<T,K>(Data<T>[] outputs, Data<K>[] inputs);

/* Implements data-parallel operations */
```

```
public class DataParallel<T,K>
{
  /* Number of cores on the underlying architecture */
  protected int _cores;

  /* Instance of the class whom method will execute either in a Map o
     Reduce */
  protected Object _currInstance;

  public DataParallel(Object ob) {
     _currInstance = ob;
     _cores = System.Environment.ProcessorCount;
  }

  /*
    Map spawns a thread for each processor core. Each thread executes
    the same method, called jobMethod, taking a different part of the
    input data, called inputTh, and returning results in a different
    part of the output data, called outputTh.
  */
  public virtual void Map( string method_name,
                           Data<T>[] outputs,
                           Data<K>[] inputs ) {

      1.  DPD<T,K> jobMethod = (DPD<T,K>)Delegate.CreateDelegate(typeof
          (DPD<T,K>), _currInstance, method_name);
      2.  Scatter input and output parameters between _cores threads,
          inputTh and outputTh
      3.  For each thread do
      3.1     jobMethod(outputTh, inputTh);
      4.  Gather output results
  }

  /*
    Reduce spawns a thread for each processor core. Each thread
    executes the same method, called jobMethod, taking a different
    part of the input data, called inputTh, and returning a single
    result applying a DPoper operation on results from threads.
  */
  public virtual void Reduce( string method_name,
                              DPoper operation,
                              Data<T>[] outputs,
                              Data<K>[] inputs ) {

      1.  DPD<T,K> jobMethod = (DPD<T,K>)Delegate.CreateDelegate(typeof
          (DPD<T,K>), _currInstance, method_name);
      2.  Scatter input and output parameters between _cores threads,
          inputTh and outputTh
      3.  For each thread do
      3.1     jobMethod(operation, outputTh, inputTh);
      4.  Gather output result
```

```
        5.  Applies operation on gathered results.
    }
}
```

Listing 2.9: C# implementation of data-parallel operations on PRAM architecture.

The constructor takes as input an object whose method, *m*, will be called either in *Map* or *Reduce*. Through .NET reflection capabilities, having both an object (e.g. *_currInstance*) and the name of one of its methods (e.g. *method_name*), it is possible to create a delegate of such method:

```
DPD<T,K> jobMethod = (DPD<T,K>)Delegate.CreateDelegate(typeof(DPD<T,K>)
    , _currInstance, method_name);
```

and then invoke the method:

```
jobMethod(outputTh, inputTh);
```

The number of cores available on the current underlying architecture is taken as the maximum degree of parallelism for a PRAM. For each core, a thread is instantiated and will execute *jobMethod*. This method's parameters are two arrays, *outputTh* and *inputTh*, which contain the partition of *outputs* and *inputs* values assigned to the current thread. The *Reduce* method also takes a value of the *DPOper* enumeration as input, that specifies what operation of reduction must be applied on input elements.

A case study of our *DataParallel* class can be vector sum, that is implemented in the following C# listing:

```
/* Library class that implements a \kernel method, named Add, for
    Vector sum */
public class MyClass {
  //...

  /* Annotated for parallel execution */
  [Kernel]
  public void Add(Data<float>[] outputs, Data<float>[] inputs) {
      outputs[0].Current = inputs[0].Current + inputs[1].Current;
  }
}

/* General application */
public class Program {
  //...

  static void Main(string[] args)
  {
      MyClass mc = new MyClass();
      float[] A;
      float[] B;
      float[] C;
```

```
    /* Vectors initialization */
    Data<float>[] outputs = new Data<float>[1];
    outputs[0] = new Data<float>(C);
    Data<float>[] inputs = new Data<float>[2];
    inputs[0] = new Data<float>(A);
    inputs[1] = new Data<float>(B);

    /* DataParallel computation initialization */
    DataParallel<float,float> dp = new DataParallel<float,float>(mc);

    /* Map function invocation having Kernel method ``Add'' as input
       */
    dp.Map("Add", outputs, inputs);
}
```

In *MyClass* method *Add* is defined with annotation *Kernel*. It must implement what is the task of a thread, as for programming languages like CUDA, Brook, CAL, etc.

2.5.5 Expressing the H-PRAM on the CLI

In this model the same consideration made for PRAM are valid except for the possibility of having distinct sub-PRAM. Since each sub-PRAM can be different from the others, they can be represented by distinct classes on the CLI, thus distinct AppDomain. Therefore, for each class a distinct variables bindings is required.
Let $\mathbb{ASMBL}$ be the domain of .NET assembly, we introduce the following function

$$partition : \mathbb{ASMBL}_{CLI} \rightarrow \mathbb{CS}_{CLI}$$

that returns a set of classes defined in a given assembly $a \in \mathbb{ASMBL}_{CLI}$. All other domains, set and functions already introduced in the PRAM mapping are still valid for the H-PRAM model. The mapping function for the H-PRAM model that, given an assembly $a \in \mathbb{ASMBL}_{\mathbb{CLI}}$ maps every variable names referred in each method $m \in met_{CLI}(c)$ for each class $c \in partition(a)$ to memory resources in the H-PRAM memory model, is defined as follows:

$$\mu_{H-PRAM}(a) = \bigcup_{c \in partition(a)} \mu_{PRAM}(c)$$

$$\mu_{H-PRAM} : ID \rightarrow MEM_{H-PRAM}$$

Considering the annotation *Kernel* introduced in the previous section, $\mu_{HPRAM}(a)$ is modified as follows:

$$\mu'_{H-PRAM}(a) = \bigcup_{c \in partition(a)} \mu'_{PRAM}(c)$$

2.5.6 Expressing the LogP on the CLI

Essentially LogP is a set of RAM that exchange messages. Therefore, actually LogP memory model extends the RAM one only with resources for messages. Under .NET remoting a message is a remotable class *rc* that inherits from `System.Runtime.-Remoting.MarshalByRefObject`. *rc* is called “remotable” because its public methods can be invoked by other remote objects that reside in other AppDomains.

Let $\mathbb{RMC}_{CLI}$ be the domain of remotable class instance name, we define the following functions:

- $rcls : \mathbb{MET}_{CLI} \rightarrow \mathbb{RMC}_{CLI}$
  as the function that given a method $m \in \mathbb{MET}_{CLI}$ returns the set of remotable class instances referred in m.
- $\varphi : \mathbb{RMC}_{CLI} \rightarrow \mathbb{COM}_{LogP}$
  it binds remotable class instances to communication unit $\in \mathbb{COM}_{LogP}$. The idea is that those units are used only for transferring data in a distributed system by the current executor.

The mapping function for the LogP model that, given a class $c \in T$ maps every variable names referred in each method $m \in met_{CLI}(c)$ for each class c to memory resources, and every remotable class instances $rc \in rcls_{CLI}(m)$ to communication resources, is defined as follows:

$$\mu_{LogP}(c) = \mu_{RAM}(c) \cup (\bigcup_{m \in rcls_{CLI}(c)} \varphi(m))$$

$$\mu_{LogP} : ID \rightarrow MEM_{LogP}$$

2.5.7 Expressing the GPGPUs model on the CLI

There are some differences in the AMD and Nvidia implementations of the computational model and memory resources design and management. However, the following definitions are general enough to be valid for all GPGPUs.

Let P_{GPGPU} be a program runnable on a GPGPU. Its behavior is described by a set of functions, written in a language targeting a GPGPU, contained by a *module*, that is a dynamically loadable package of device code and data, akin to DLLs in Windows. Let $\mathbb{FUN}_{GPGPU}$ be the domain of the functions, $\mathbb{MD}_{GPGPU}$ the domain of the modules, we define:

$$func_{GPGPU} : \mathbb{MD}_{GPGPU} \rightarrow \mathbb{FUN}_{GPGPU}$$

as the function that given a GPGPU module $mod \in MD_{GPGPU}$ returns a set of all the functions $f_i \in \mathbb{FUN}_{GPGPU}$ declared inside it.

$$\eta_{GPGPU} : \mathbb{FUN}_{GPGPU} \rightarrow \mathbb{IL}_{GPGPU}$$

as the function that given a function $f \in \mathbb{FUN}_{GPGPU}$ returns its list of instructions.

Considering state spaces introduced in Section 1.4.2 we define these domains. Let $\mathbb{RG}_{GPGPU}$ be the domain of registers in the register state space, $\mathbb{LM}_{GPGPU}$ the domain of the local memory unit in the local state space, $\mathbb{SM}_{GPGPU}$ the domain of the shared memory unit in the shared state space, $\mathbb{GM}_{GPGPU}$ the domain of the global memory units in the global state space. We define the domain of memory resources as:

$$\mathbb{MEM}_{GPGPU} = \mathbb{RG}_{GPGPU} \ \cup \ \mathbb{LM}_{GPGPU} \ \cup \ \mathbb{SM}_{GPGPU} \ \cup \ \mathbb{GM}_{GPGPU}$$

Moreover, we introduce $\mathbb{COMC}_{GPGPU}$ the domain of the communication units, that are input/output streams on GPGPUs.

Finally, a program that runs on a GPGPU is formalized by the following tuple:

$$P_{GPGPU} \equiv \mathbb{FUN}_{GPGPU} \times \mathbb{MEM}_{GPGPU} \times GPGPU \times \mathbb{COMC}_{GPGPU}$$

2.5.8 Formal definition of the CIL to GPGPU compiler

In this section we define formally the mapping function μ_{GPGPU} that applied to classes in a program P'_{CLI} as input returns an equivalent program P_{GPGPU}. Moreover, we describe how the functions introduced in Section 2.5.4 must be changed for a feasible mapping. In the following section we prove the correctness of μ_{GPGPU}, along with the definition of *equivalence* between P'_{CLI} and P_{GPGPU}.

Given a class $c \in T$

$$\mu_{GPGPU}(c) = \bigcup_{m \in met_{CLI}(c)} \left(\begin{array}{l} \rho(mlocals_{CLI}(m)) \ \cup \ \omega(params_{CLI}(m)) \\ \cup \ \lambda(ifields_{CLI}(m)) \ \cup \ \sigma(sfields_{CLI}(m)) \end{array} \right)$$

$$\mu_{GPGPU} : ID \rightarrow MEM_{GPGPU}$$

for each method $m \in met_{CLI}(c)$, μ_{GPGPU} maps each variable $v \in \mathbb{MEM}_{CLI}$, that is referred in m to $\mathbb{MEM}_{GPGPU}$, i.e. a memory resource in the GPGPU memory model (recall Section 1.4.2).

To guarantee a feasible mapping it is important to consider that due to GPGPUs architectural constraints, only a sub-set of types in the CLI Common Type System (CTS) can be mapped on GPGPUs types, as shown in table 2.1.

The definition of μ_{GPGPU} leverages on functions introduced in Section 2.5.4. However, they must be changed to consider the GPGPU memory model as follows:

PTX Basic Type		Name in CIL
Signed integer	.s8	int8
	.s16	int16
	.s32	int32
	.s64	int64
Unsigned integer	.u8	unsigned int8
	.u16	unsigned int16
	.u32	unsigned int32
	.u64	unsigned int64
Floating-point	.f32	float32
	.f64	float64
Predicate	.pred	bool

Table 2.1: PTX and CIL types mapping.

- $\rho : \mathbb{LOC}_{CLI} \rightarrow \mathbb{MEM}_{GPGPU}$ it binds method's local variable names to a register $\in \mathbb{RG}_{GPGPU}$ of $\mathbb{MEM}_{GPGPU}$ with the same type.
- $\omega : \mathbb{MPRM}_{CLI} \rightarrow \mathbb{COM}_{GPGPU}\}$ it binds method's formal parameter names to special registers $\in \mathbb{COM}_{GPGPU}$ in the parameter state space, used by GPGPUs to communicate with the CPU.
- $\lambda : \mathbb{IFLD}_{CLI} \rightarrow \mathbb{MEM}_{GPGPU}$ it binds instance field names to local memory unit $\in \mathbb{LM}_{CLI}$ of $\mathbb{MEM}_{GPGPU}$. These variables can be accessed only by a thread running an instance of c.
- $\sigma : \mathbb{SFLD}_{CLI} \rightarrow \mathbb{MEM}_{GPGPU}$ it binds static field names to memory units $\in \mathbb{GM}_{GPGPU}$ of $\mathbb{MEM}_{GPGPU}$.

Other functions, such as met_{CLI}, η_{CLI}, $mlocals_{CLI}$, $params_{CLI}$, $ifields_{CLI}$ and $sfields_{CLI}$, are all provided by the CLI reflection capabilities in `System.Reflection` namespace.

Chapter 3

Parallelism exploitation

In the past many different parallel programming languages have been introduced, all based on the following conjecture: an effective (high performance, machine independent) parallel programming language cannot be founded on the standard Von Neumann model of sequential computation, rather must be founded on an abstract parallel machine that reflects characteristics of real parallel architectures.

The advantage of writing sequential programs is that the code is easier to debug and optimize for full speed execution than parallel programs. As a result, many programming languages inherit this single flow of control that forces sequential programming. However, in order to benefit from rapidly improving computer performance and to retain the "write once, run faster on new hardware" paradigm, commercial and scientific software must switch to new software development and system support mechanisms [100] and the latter must enable a significant portion of the programming community to construct parallel applications.

While the architecture community has been addressing the need for more computing power, techniques for parallel programming have been advancing relatively slowly. Programming paradigms that emerged in the 1980s and 1990s, such as message passing, are still popular. Although significant advances have been made in raising the level of abstraction in parallel programming [41, 27, 26], parallel compilers [101], and program optimization tools [102], parallel programming has remained the province of a small number of specialists. Determining how to program a parallel processor efficiently is a difficult task that requires the programmer to understand many details about the computer architecture and parallel algorithms. Computer scientists have been developing various techniques for both detecting and utilizing parallelism.

This chapter surveys those approaches to parallel programming that have inspired **4-Centauri**'s design and implementation.

3.1 Why is parallel programming so difficult?

There are a number of reasons for that. First, decomposing, or mapping, data structures and tasks in a sequential program into parallel parts is a challenge. Most programming languages are mainly sequential, making unnatural to tackle parallelism in the first place.
Second, writing parallel code that runs correctly requires synchronization and coordination among processors.
Third, debugging parallel applications can be a challenging task [103]. The increased complexity of multi-threaded programs results in a large number of possible states that the program may be in at any given time. Determining the state of the program at the time of failure can be difficult; understanding why a particular state is troublesome can be even more difficult. Parallel programs often fail in unexpected ways, and often in a nondeterministic fashion. Bugs may manifest themselves in a sporadic fashion, frustrating developers who are accustomed to troubleshooting issues that are consistently reproducible and predictable. Moreover, parallel applications can fail in a drastic fashion-deadlocks cause an application or worse yet, the entire system, to hang. Users tend to find these types of failures to be unacceptable.
Finally, the most difficult task is writing efficient parallel programs, a task that requires careful balancing between the program's communication and computation parts. When dealing with a single-processor architecture, the programmers try to minimize the number of operations the processor must perform. Counting the number of operations in a sequential application is generally straightforward, and well-understood performance bounds exist for many algorithms. When the program is moved to a parallel machine, minimizing the number of computations does not guarantee optimal performance. The same is true of sequential algorithms on today's complex processors because of multi-level memory hierarchies. The problem is heightened when moving to a parallel system. First, the theoretical speedup of the entire program is limited by the longest serial path, as quantified by Amdahls law [104]. Second, the speedup of parallelizable part of the program is highly dependent on interprocessor communication. Efficiently mapping the application, or distributing parts of the application between multiple processing elements, becomes increasingly important. The performance bounds of parallel algorithms depend strongly on how the data and computation are distributed or mapped; this dependence makes it more difficult to estimate and optimize performance. The situation is further complicated because algorithmic details might change as the algorithm is moved from a single processor to a multiprocessor machine. Often the most efficient sequential algorithm is not the most efficient parallel one.

3.2 Existing approaches to parallel programming

The solution of designing a successful parallel programming paradigm can be divided into three cases:

1. Use an existing sequential paradigm, letting the compiler to deal with concurrent execution.

2. Augment an existing sequential paradigm, providing information about concurrency.

3. Create a new paradigm.

The first approach preserves the value of legacy codes, but automatic parallelization is a very difficult task for which little progress has been made in a decade and a half. Both the second and third approaches devalue the existing software legacy, either requiring that existing programs be revised to incorporate the parallel constructs, perhaps at considerable intellectual effort, or that programs be completely rewritten. There is no effortless solution: the second approach may introduce constructs that conflict with sequential semantics, while the third approach requires that programmers learn another language and re-implement their existing code. In all cases the greatest challenge is in enabling programmers to write machine-independent programs despite the wide variability of parallel platforms. Clearly, abstractions are needed to raise the level of expression away from the hardware, but abstractions that prevent compilers from exploiting the best hardware features will degrade performance greatly.

In all approaches a set of design decisions must be taken. The first decision is about the optimization, that is, whether the approach is meant to optimize a program for a sequential or a parallel architecture. In the former case, the approach entails finding an efficient mapping into the memory hierarchy of a single-processor machine, such as determining the optimal strategy for cache utilization. In the latter case, the approach needs to optimize the code for a distributed memory. The second design decision is about the support layer, that is, the software layer in which the (automatic) distribution and optimization is implemented. Optimization approaches tend to be implemented in either the compiler or middleware layer. If the parallelization approach is implemented in the compiler, it does not have access to run-time information that could significantly influence the chosen mapping. On the other hand, if the approach is implemented in the middleware and is invoked at run-time, it could incur a significant overhead because of the extra effort required to collect the run-time information. The third design decision is about code analysis, that is, how the approach finds instances of parallelism. Code analysis can be static or dynamic. Static code analysis involves looking at the code as text and trying to extract inherent parallelism on the basis of how the program is written. Dynamic code analysis involves analyzing the behavior of the code as it is running, thus

allowing access to run-time information. The fourth design decision is at what scope the approach applies optimizations. Approaches can be local (peephole) or global (program flow). Local optimization approaches, which find optimal distributions for individual functions, have had the most success and are used by many parallel programmers.

In the following sections we illustrate the main features of those languages and libraries we have considered for designing and implementing **4-Centauri**.

3.2.1 Automatic program parallelization

Many attempts in the past have been done to design and implement a compiler able to auto-parallelize any code provided as input [22] However, most attempts failed except for some execution patterns, such as loops.

ADAPTOR

An example of automatic data-parallelism compiler is ADAPTOR (Automatic DAta Parallelism TranslatOR) [23]. ADAPTOR is a public domain High Performance Fortran (HPF) [24] and OpenMP [39] Fortran compilation system. It translates data parallel HPF programs into parallel programs using process parallelism with message passing for distributed memory architectures or thread parallelism with synchronization for shared memory architectures. Process and thread parallelism can be utilized together on clusters of shared memory systems. Furthermore, ADAPTOR compiles OpenMP Fortran programs for shared memory system by exploiting thread parallelism. The ADAPTOR compilation system package consists of:

- the source to source transformation tool,
- the distributed array library which is the HPF and OpenMP run-time system that handles descriptors for arrays, sections and distributions and handles communication and synchronization routines,
- the compiler driver.

A HPF program is compiled into an equivalent parallel program based on MPI [31] and/or PThreads [105]. The compiler driver adaptor invokes the source-to-source translation that generates a parallel program with explicit thread or MPI parallelism from the HPF program. Afterwards, it invokes a native Fortran 77 or Fortran 90 compiler to compile the generated code. Finally, the compiled codes are linked with the LIBADP run-time system and the utilized MPI and/or Pthreads library. While the message passing programs are well suited for distributed memory (DM) architectures, the parallel programs based on PThreads run well on shared memory (SM) architectures. In a similar way, HPF programs can also be compiled for sequential machines by ignoring the parallelism, and also for clusters of shared memory architectures by exploiting nested process and thread parallelism.

SUIF

The SUIF (Stanford University Intermediate Format) [25] system is a compiler infrastructure designed to support collaborative research and development of compilation techniques, based upon a program representation. It has a modular architecture that comprises a small *Kernel* which implements a set of basic functions found to be useful across all compilation passes, a number of modules loaded dynamically under developers control, and a driver that controls the system operation. The SUIF *Kernel* defines and implements the compiler environment that is all the user needs to know when writing a SUIF program. The bulk of the SUIF compiler system is structured as modules, each of which is a C++ class identified by a unique name.

The SUIF design supports high-level program analysis of C and Fortran programs. The SUIF key features are:

- modular subsystem that allows different components to be combined easily.
- extensible program representation that allows users to create new instructions to capture new program construct semantics or new program analysis concepts.There is a predefined object hierarchy to capture the program semantics, and developers are able to refine these abstractions for their needs.

pMapper

pMapper is an automatic mapping engine originally designed to distribute MATLAB programs onto parallel computers, specifically clusters. A map can be defined with three pieces of information: grid specification, distribution description, and processor list. Since the task of mapping the program is separated from the task of developing the algorithm, the entity that determines the maps for the program could be another layer of software, the pMapper.

The pMapper framework globally optimizes performance of parallel programs at run-time. To do this, pMapper requires a presence of an underlying parallel library. For more details see [26]. This library has increased the level of abstraction by implementing a map layer that insulates the algorithm developer from writing complicated message-passing code. These libraries introduce the concept of map independence, that is, the task of mapping the program onto a processing architecture is independent from the task of algorithm development. Once the algorithm has been specified, the user can simply define maps for the program without having to change the high-level algorithm. The maps can be changed without having to change any of the program details. The key idea behind map independence is that a parallel programming expert can define the maps, while a domain expert can specify the algorithm. To provide accurate mappings, we need to collect benchmark performance data of the parallel library on the target parallel architecture.

The task of benchmarking the library is computationally intensive, making it infeasible to collect sufficient timing data during program execution. Once the

benchmarking data have been collected, pMapper uses them to generate maps in an efficient manner. This process naturally yields a two-phase mapping architecture. The initialization phase occurs once, when pMapper is installed on the target architecture or, if the architecture is simulated, when the architecture parameters are first specified. Once the timing data are collected and stored as a performance model, they are used to generate mappings.

The mapping and execution phase is performed once for each program at run-time. pMapper uses lazy evaluation, that is, it delays execution until necessary. This approach allows pMapper to have the greatest possible amount of information about the program to be mapped at mapping time.

3.2.2 New programming language paradigms

ZPL

ZPL [27] is a new programming language that is especially effective for scientific and engineering computations. It is intended to replace languages such as Fortran and C for technical computing. It is:

- an array language. Expressions such as $X + Y$ have been generalized to apply to whole arrays as well as simple scalars, depending on how X and Y are declared. Not only does ZPL save the programmer from writing many tedious loops and specifying error prone index calculations, it enables the compiler to identify parallelism that will speed the computation.

- machine independent programming language, meaning that ZPL programs run well on both sequential and parallel computers. Programmers need not concern themselves with machine specifics.

- implicitly parallel programming language. That is, although ZPL was designed to simplify programming parallel computers, programmers do not specify how the computation is performed concurrently. Nor do they insert interprocessor communication. The ZPL compiler is responsible for producing parallel object code from the source program, and for taking care of all details necessary to exploit the target parallel computer.

- a global-view parallel language with communication cues. The programmer writes code that largely disregards the processors that will execute it. Programmers do not write interprocessor communication commands. The details of communication are managed by the compiler, but the programmer is readily aware of where communication is induced. This provides a simple, but powerful performance model called the what-you-see-is-what-you-get (WYSIWYG) performance model.

Google Map-Reduce

MapReduce [28] is a programming model and an associated implementation for processing and generating large data sets. A developer expresses the computation as two functions: Map and Reduce. Map takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function. The Reduce function accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

When the user program calls the MapReduce function, the following sequence of actions occurs:

1. Input data is split into M map tasks.
2. Many copies of the program start up on a cluster of machines. One of the copies of the program is special, called the master. The rest are workers that are assigned work by the master.
3. Reduce phase partitioned into R reduce tasks.
4. Master assigns each map task to a free worker
   (a) Tasks are assigned to workers dynamically, considering locality of data to worker when assigning task.
   (b) Worker reads task input (either from memory or local disk.
   (c) Worker produces R local files containing intermediate (k,v) pairs
5. Master assigns each reduce task to a free worker
   (a) Worker reads intermediate (k,v) pairs from map workers.
   (b) Worker sorts and applies users Reduce operation to produce the output.
   (c) User may specify Partition: which intermediate keys to which Reducers.
6. MapReduce library gathers together all pairs with the same key (shuffle/sort)

Programs written are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication.

Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may

be suitable for a small shared-memory machine, another for a large NUMA[1] multiprocessor, and yet another for an even larger collection of networked machines.

X10

X10 [29] is a statically typed object-oriented language, extending the Java sequential core language with places, activities, clocks, (distributed, multi-dimensional) arrays and struct types. An X10 program is intended to run on a wide range of computers, from uniprocessors to large clusters of parallel processors supporting millions of concurrent operations. To support this scale, X10 introduces the central concept of place.

Conceptually, a place is a “virtual shared-memory multi-processor”: a computational unit with a finite number of hardware threads and a bounded amount of shared memory, uniformly accessible by all threads. An X10 computation acts on data objects through the execution of lightweight threads called activities. Objects are of two kinds. A scalar object has a small, statically fixed set of fields, each of which has a distinct name. A scalar object is located at a single place and stays at that place throughout its lifetime. An aggregate object has many fields uniformly accessed through an index and may be distributed across many places. The distribution of an aggregate object remains unchanged throughout the computation. X10 assumes an underlying garbage collector will dispose of objects and reclaim the memory associated with them once it can be determined that these objects are no longer accessible from the current state of the computation.

X10 has a unified or global address space. This means that an activity can reference objects at other places. However, an activity may synchronously access data items only in the current place (the place in which the activity is running). It may atomically update one or more data items, but only in the current place. To read a remote location, an activity must spawn another activity asynchronously. This operation returns immediately, leaving the spawning activity with a future for the result. Similarly, remote location can be written into only by asynchronously spawning an activity to run at that location. Throughout its lifetime an activity executes at the same place. An activity may dynamically spawn activities in the current or remote places.

X10 provides multiple barriers in a dynamic context, called clocks, while still supporting determinate, deadlock-free parallel computation. Activities may use clocks to repeatedly detect quiescence of arbitrary programmer-specified, data-dependent set of activities. Each activity is spawned with a known set of clocks and may dynamically create new clocks. At any given time an activity is registered with zero or more clocks. At any given step of the execution a clock is in a given phase. It advances to the next phase only when all its registered activities have quiesced (by executing a next operation on the clock). When a clock advances, all its activities

[1] Not Uniform Memory Access

may now resume execution. Thus clocks act as barriers for a dynamically varying collection of activities.

X10 supports annotations on classes and interfaces, methods and constructors, variables, types, expressions and statements. These annotations may be processed by compiler plugins.

Fortress

The Fortress [30] is a general-purpose, statically typed, component-based programming language. It is mainly an object-oriented programming language, but also provides support for a functional style. The Fortress type system is based on *traits*, a feature similar to interfaces in the Java programming language, except that *traits* can contain code (but no fields). Its type system integrates nominal subtyping, generic types using "where" clauses, and it retains type information at run time to allow type-dependent operations on generic types.

Fortress is implicitly parallel and provides constructs and annotations to serialize execution when necessary. Even if it allows to specify when evaluation may proceed in parallel (or rather, when it must not), Fortress does not require it to do so. As a result, a compiler or virtual execution environment need to concern itself with determining whether executing a program in parallel is advantageous or not. If there are no spare processors, for example, or distributing the computation is more expensive than doing it, or for any other reason, an implementation may execute parallel tasks sequentially, and in any order. Fortress implements a technique called *work stealing*, such that its runtime support can effectively distribute parallel tasks to multiple processors with minimal overhead.

Since Fortress' design is not tailored to a specific architecture, it must parallel execution of tasks accessing shared and distributed memory. In the former case, Fortress avoids race conditions through the support for atomic expressions[2]. In the latter case, the cost of access to memory could be extremely nonuniform, so it could be crucial to place data "near" to the processor that uses it. To manage this, Fortress provides distributions, which specify "regions" of the machine where data resides and computation occurs. These regions are arranged in a hierarchy that abstractly represents the relative cost of access: computation can cheaply access data in its own or nearby regions.

3.2.3 Extending standard design

The problem with fully automatic schemes is that they are best suited for detecting small grain parallelism, whereas the complexity of schemes in which the programmer is completely responsible for managing the parallel environment can overwhelm the programmer.

[2]An atomic expression appears to be executed in isolation: the evaluation of an atomic expression never appears to be interleaved with operations due to other tasks.

Mentat

The Mentat [40] philosophy on parallel computing is guided by two observations. First, that the programmer understands the problem domain of the application and can make better data and computation partitioning decisions than compilers can. The truth of this is evidenced by the fact that most successfully produced parallel applications have been hand-coded using low-level primitives. In these applications the programmer has decomposed and distributed both the data and computation. Second, the management of tens to thousands of asynchronous tasks, where errors dependent on timing are easy to make, is beyond the capacity of most programmers unless a tremendous amount of effort is expended. The truth of this is evidenced by the fact that writing parallel applications is almost universally acknowledged to be far more difficult than writing sequential applications. Compilers, on the other hand, are very good at ensuring that events happen in the right order, and can more readily and correctly manage communication and synchronization, particularly in highly asynchronous, non-SIMD, environments.

There are two primary components of Mentat: the Mentat Programming Language (MPL) and the Legion run-time system. MPL is an object-oriented programming language based on C++ that masks the difficulty of the parallel environment from the programmer. The granule of computation is the Mentat class instance, which consists of contained objects (local and member variables), their procedures, and a thread of control. Programmers are responsible for identifying those object classes that are of sufficient computational complexity to allow efficient parallel execution. Instances of Mentat classes are used just like ordinary C++ classes, freeing the programmer to concentrate on the algorithm, not on managing the environment. The data and control dependencies between Mentat class instances involved in invocation, communication, and synchronization are automatically detected and managed by the compiler and run-time system without further programmer intervention. By splitting the responsibility between the compiler and the programmer we exploit the strengths of each, and avoid their weaknesses. The underlying assumption is that the programmer can make better granularity and partitioning decisions, while the compiler can correctly manage synchronization. This simplifies the task of writing parallel programs, making the power of parallel and distributed systems more accessible.

MPL is an extended C++ designed to simplify the task of writing parallel applications by providing parallelism encapsulation. Parallelism encapsulation takes two forms, intra-object encapsulation and inter-object encapsulation. In intra-object encapsulation of parallelism, callers of a Mentat object member function are unaware of whether the implementation of the member function is sequential or parallel, i.e., whether its program graph is a single node or a parallel graph. In inter-object encapsulation of parallelism, programmers of code fragments (e.g., a Mentat object member function) need not concern themselves with the parallel execution opportunities between the different Mentat object member functions they invoke.

Mentat objects are independent objects. Independent objects are analogous to UNIX processes. Mentat uses an object model that distinguishes between two types of objects, contained objects and independent objects. Contained objects are objects contained in another objects address space. Instances of C++ classes, integers, structures, and so on, are contained objects. Independent objects possess a distinct address space, a system-wide unique name, and a thread of control. Communication between independent objects is accomplished via member function invocation.

Rapid Mind

The RapidMind Development Platform (RDP) [42] is a framework for expressing data-parallel computations from within C++ and executing them efficiently on multicore processors. It lets to specify any computation that can leverage multiple cores within existing C++ applications. In particular, the main goal is to give developers access to the power of both the Cell BE and the GPU, but at a high level, so they can focus on developing efficient parallel algorithms instead of managing low-level, architecture-specific details. The RDP provides dynamic code generation for multiple specialized hardware targets.

RDP has a C++ interface for describing computation, rather than a separate language. The platform interface is implemented using only ISO standard C++ features and works with any ISO standard C++ compiler. It can be used just by including a header file and linking to a library. The platform interface permits the expression of arbitrary computation. In fact, the system includes a staged, dynamic compiler supporting run-time code generation. This enables some novel high-performance programming techniques above and beyond the exploitation of parallelism.

RDP includes an extensive run-time component as well as interface and dynamic compilation components. The run-time component automates common tasks such as task queuing, data streaming, data transfer, synchronization, and load-balancing. It asynchronously manages tasks executing on remote processors and manages data transfers to and from distributed memory. This run-time component provides a framework for efficient parallel execution of the computation specified by the main program.

The platform uses a bulk-synchronization model that supports a conceptual single thread of control, making debugging straightforward. The structure of the language makes parallelism explicit, however, encouraging the development and use of efficient and scalable parallel algorithms.

Unlike alternatives such as OpenMP [39], MPI, and threads, their parallel programming model is portable to a wide range of parallel hardware architectures, including vector and stream machines, such as GPUs, as well as distributed memory machines, such as the Cell BE. The system provides a strong execution and data abstraction that is simultaneously modular, portable, and efficient.

Users of the RDP continue to program in C++ using their existing compiler.

After identifying components of their application to accelerate, the overall process of integration is as follows:

1. Replace types: The developer replaces numerical types representing floating point numbers and integers with the equivalent RapidMind platform types.

2. Capture computations: While the users application is running, sequences of numerical operations invoked by the users application are captured, recorded, and dynamically compiled to a program object by the RapidMind platform.

3. Stream execution: The RapidMind platform run-time is used to manage parallel execution of program objects on the target hardware (in this case, a GPU)

The RDP supports two execution mode: immediate mode, operations can be executed on the host processor, and retained mode, operations are recorded and dynamically compiled into a "program object". These program objects can be used as functions in the host program. In the case of GPUs, applying such a function to an array of values automatically invokes a massively parallel computation on the video accelerator. Data is automatically transferred to and from the video accelerator, overlapping computation with data transfers.

OpenMP

OpenMP [39] is an API for writing shared memory parallel applications in C, C++, and Fortran. It consists of compiler directives, run-time routines and environment variables.

The OpenMP API uses the fork-join model of parallel execution. Multiple threads of execution perform tasks defined implicitly or explicitly by OpenMP directives. OpenMP is intended to support programs that will execute correctly both as parallel programs (multiple threads of execution and a full OpenMP support library) and as sequential programs (directives ignored and a simple OpenMP stubs library). However, it is possible and permitted to develop a program that executes correctly as a parallel program but not as a sequential program, or that produces different results when executed as a parallel program compared to when it is executed as a sequential program.

An OpenMP program begins as a single thread of execution, called the initial thread. The initial thread executes sequentially, as if enclosed in an implicit task region, called the initial task region, that is defined by an implicit inactive parallel region surrounding the whole program.

The code for each task is defined by the code inside the parallel construct. Each task is assigned to a different thread in the team and becomes tied; that is, it is always executed by the thread to which it is initially assigned. The task region of the task being executed by the encountering thread is suspended, and each member of the new team executes its implicit task. There is an implicit barrier at the

end of the parallel construct. Beyond the end of the parallel construct, only the master thread resumes execution, by resuming the task region that was suspended upon encountering the parallel construct. Any number of parallel constructs can be specified in a single program.

OpenMP provides a relaxed-consistency, shared-memory model. All OpenMP threads have access to a place to store and to retrieve variables, called the memory. In addition, each thread is allowed to have its own temporary view of the memory. The temporary view of memory for each thread is not a required part of the OpenMP memory model, but can represent any kind of intervening structure, such as machine registers, cache, or other local storage, between the thread and the memory. The temporary view of memory allows the thread to cache variables and thereby to avoid going to memory for every reference to a variable. Each thread also has access to another type of memory that must not be accessed by other threads, called thread private memory.

The memory model has relaxed-consistency because a threads temporary view of memory is not required to be consistent with memory at all times. A value written to a variable can remain in the threads temporary view until it is forced to memory at a later time. The OpenMP flush operation enforces consistency between the temporary view and memory. If a thread has performed a write to its temporary view of a shared variable since its last flush of that variable, then when it executes another flush of the variable, the flush does not complete until the value of the variable has been written to the variable in memory. The flush operation provides a guarantee of consistency between a threads temporary view and memory.

Muskel

Muskel [41] is a full Java skeleton programming environment that provides stateless task farm and pipeline skeletons. These skeletons can be arbitrarily nested, to program pipelines with farm stages, for example, and they process a single stream of input tasks to produce a single stream of output tasks. Muskel implements skeletons using data flow technology and Java RMI facilities. The programmer using Muskel can express parallel computations by simply using the provided Pipeline and Farm classes. In order to execute the program, developers set up a Manager object. Then, using appropriate methods, they indicate to the manager the program to execute, the performance contract required (in this case, the parallelism degree required for the execution), what is in charge of providing the input data (the input stream manager, which is basically an iterator providing the classical boolean hasNext() and Object next() methods) and what is in charge of processing the output data (the output stream manager, providing only a void deliver (Object) method processing a single result of the program). Finally, they can request parallel program execution simply by issuing an *eval* call to the manager. When the call terminates, the output file has been produced.

Actually, the *eval* method execution happens in steps. First, the application

manager looks for available processing elements using a simplified, multicast based peer-to-peer discovery protocol, and recruits the required remote processing elements. Each remote processing element runs a data flow interpreter. Then the skeleton program (the main of the example) is compiled into a macro data flow graph and a thread is forked for each of the remote processing elements recruited. Then the input stream is read. For each task item, an instance of the macro data flow graph is created and the task item token is stored in the proper place (initial data flow instruction(s)). The graph is placed in the task pool, the repository for data flow instructions to be executed. Each thread looks for a fireable instruction in the task pool and delivers it for execution to the associated remote data flow interpreter. The remote interpreter instance associated to the thread is initialized by being sent the serialized code of the data flow instructions, once and for all, before the computation actually starts. Once the remote interpreter terminates the execution of the data flow instruction, the thread either stores the result token in the appropriate "next" data flow instruction(s) in the task pool, or it directly writes the result to the output stream, invoking the deliver method of the output stream manager. Currently, the task pool is a centralized one, associated with the centralized manager.

The manager takes care of ensuring that the performance contract is satisfied. The policies implemented by the Muskel managers are best effort. The Muskel library tries to do its best to accomplish user requests. If it is not possible to completely satisfy the user requests, the library establishes the closest configuration to the one implicitly specified by the user with the performance contract. In the current version of the Muskel prototype, the only performance contract actually implemented is the ParDegree one, asking for the use of a constant number of remote interpreters in the execution of the program.

3.2.4 GPGPU programming languages

GPGPUs, coupled with recent improvements in their programmability, have become a compelling platform for computationally demanding tasks in a wide variety of application domains. They are especially well-suited to address problems that can be expressed as data-parallel computations (i.e. the same program is executed on many data elements in parallel) with high arithmetic intensity (i.e. the ratio of arithmetic operations to memory operations). Because the same program is executed for each data element, there is a lower requirement for a sophisticated flow control; and because it is executed on many data elements and has high arithmetic intensity, memory access latency can be hidden with calculations instead of using big data caches. Many applications that process large data sets can use a data-parallel programming model to speed up computations: 3D rendering, interactive cinematic lighting, video encoding and decoding, image scaling, stereo vision, and pattern recognition.

Stream programming model

The latest generation of GPUs from Nvidia and AMD have already taken a significant step toward the data-parallel programming model by supporting a separate model for non-graphics computations that is more flexible than the one used for graphics, called Stream Programming. The stream programming model exposes the parallelism and communication patterns inherent in the application by structuring data into *streams* and expressing computation as *shader* (in the traditional 3D terminology) or *Kernel* (a term to stress the will to go beyond 3D graphics) that operate on streams in SIMD fashion.
A typical stream program is composed of the following steps:

1. The developers pinpoints the data-parallel section of their applications, so it is split into independent parallel sections, called *Kernels*. The input and output of each *Kernel* is one or more arrays stored in the GPU's local memory.

2. To execute a *Kernel*, the computation domain (or the size of the output stream) must be specified.

3. The rasterizer generates a fragment for every element in the domain.

4. Each of the generated fragments is then processed by the active *Kernel*'s fragment program.

5. The output of the fragment program is a value per fragment.

Both Nvidia and AMD vendors provide their stream languages (e.g. AMD Brook+ [21], AMD CAL [20] and Nvidia CUDA [19]) Currently, both AMD CAL and Nvidia CUDA programming interfaces consist of a minimal set of extensions to the C language, that allow the programmer to target portions of the source code for execution on the device, and a run-time library split into:

- *Host application*, that performs application work, and sends commands to the GPU using CAL/CUDA API.

- *GPU Kernel* , that reads input data, performs stream parallel computation and writes output data.

- *Common component*, that provides built-in vector types and a subset of the C standard library that are supported in both host and device code.

The main differences between these platforms relate to:

- *Kernel* implementation; AMD IL instructions are defined in both text and binary formats so that developers can select between easier program maintenance (text format) and execution speed (binary format) for time-critical,

run-time applications. The text instruction format resembles assembly language, whereas the binary instruction format resembles machine code. In both cases, an AMD IL compiler is required, either at compile-time or run-time, to convert the IL instructions to machine-specific codes. Nvidia allows *Kernel* coding only using CUDA (i.e. C language) thus the compiler is required at compile-time only.

- *thread hierarchy*; Nvidia CUDA divides the threads into a grid of blocks. Threads within a block can cooperate among themselves by sharing data through some shared memory, and synchronizing their execution to coordinate memory accesses. The number of threads per block is restricted by the limited memory resources of a processor core. AMD provides no control over threads configuration at run-time.

- *execution configuration*; Nvidia gives developers support on static *Kernel* execution configuration. The arguments[3] to the execution configuration (i.e. grid and blocks dimension) are evaluated before the actual execution. At run-time, it is the underlying GPU hardware in charge of schedule threads over cores.

Nvidia CUDA. CUDA is a software architecture for issuing and managing computations on the GPU as a data-parallel computing device, without the need for mapping them to a graphics API. CUDA extends C by allowing the programmer to define special C functions (*Kernels*) that, when called, are executed n times in parallel by n different CUDA threads, as opposed to only once like regular C functions. Each thread is contained in a thread block. Threads within a block can cooperate among themselves by sharing data through some shared memory, and synchronizing their execution to coordinate memory accesses. The number of threads per block is restricted by the limited memory resources of a processor core. On the Nvidia Tesla architecture, a thread block may contain up to 512 threads. However, a *Kernel* can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block multiplied by the number of blocks. Programmers can specify the number of blocks and the number of threads for each block through the first and second parameters of the *Kernel* function signature.
The CUDA software stack is composed of several layers: a device driver, an application programming interface (API) and its run-time, and two higher-level mathematical libraries of common usage, CUFFT and CUBLAS. The CUDA programming interface consists of:

- A minimal set of extensions to the C language that allow the programmer to target portions of the source code for execution on the device.

[3] grid and blocks dimension that will be used to execute the function on the device, as well as the associated stream.

- The run-time library is split into:
  - Host component, that runs on the CPU and provides functions to control and access one or more computation devices from the host.
  - Device component (*Kernel*), that runs on the device and provides device-specific functions. Device-side function can be of two types: `__global__` and `__device__`.
  - Common component, that provides built-in vector types and a subset of the C standard library that are supported in both host and device code.

CUDA introduces function type qualifiers in order to define where a function executes, which functions can call it and which ones it can invoke:

- Executable on host, callable from the host component, `__host__` typed function can invoke all function not typed by `__device__`.

- Executable on device, callable from the host component, `__global__` typed function can invoke `__device__` typed function only.

- Executable on device, callable from the device component, `__device__` typed function.

`__global__` and `__device__` typed function have several restrictions, for instance they do not support recursion, cannot declare static variables, and cannot have a variable number of arguments. `__global__` functions are asynchronous so they return before the device has completed its execution. `__global__` function must specify its execution configuration. This defines the dimension of the grid and blocks that will be used to execute the function on the device, as well as the associated stream. It is specified by inserting an expression of the form `<<<` *Dg*, *Db*, *Ns*, *S* `>>>` between the function name and the parenthesized argument list, where:

- *Dg* specifies the dimension and size of the grid, such that *Dg.x* * *Dg.y* equals the number of blocks being launched, whereas Dg.z is unused;

- *Db* specifies the dimension and size of each block, such that *Db.x* * *Db.y* * *Db.z* equals the number of threads per block;

- *Ns* specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory; this dynamically allocated memory is used by any of the variables declared as an external array; *Ns* is an optional argument which defaults to 0;

- *S* specifies the associated stream; it is an optional argument which defaults to 0.

It is important to consider the synchronization function of the Device run-time component, usable to coordinate communication between threads of a same block. It allows implementing reduction directly on the device, eventually avoiding useless GPU-CPU communication. Another important feature provided is Atomic instructions. They perform read-modify-write atomic operations on one 32-bit or 64-bit word residing in global or shared memory. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. These instructions are very important since using them make it possible to implement thread synchronization on the global memory, thus between multiple threads of different blocks.

We provide a trivial code in CUDA that implements the *saxpy* function in double precision. The *saxpy* is a combination of scalar multiplication and vector addition. In order to highlight the main differences between programming languages provided by Nvidia and AMD, and for each language the level of abstraction and main part of it, we'll present the same example in the following paragraphs. In listing 3.1, the function *saxpyCUDAKernel* implements the *saxpy* function. *blockIdx*, *blockDim* and *threadIdx* are built-in variables that return the CTA (thread block) x-dimensional index, the dimension of the CTA, and the thread index inside a CTA respectively.

```
__global__ void saxpyCUDAKernel(float a, float* InData1, float* InData2
    , float* Result) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  Result[idx] = InData1[idx] * a + InData2[idx];
}
```

Listing 3.1: CUDA version of the *saxpy* function.

In listing 3.2 we provide the code required to setup and execute the *Kernel* function implemented in listing 3.1.

```
int main(int argc, char** argv) {
  float* InitData1; float* InitData2;
  float* InData1; float* InData2;
  float* Result; float* HostResult;
  float a = 10.0;
  unsigned int Length = 100;

  /* ---------- INITIALIZATION ---------- */
  CUT_DEVICE_INIT(argc, argv);
  cudaStream_t stream;
  CUDA_SAFE_CALL( cudaStreamCreate(&stream) );

  /* ---------- MEMORY ALLOCATION ---------- */
  CUDA_SAFE_CALL( cudaMallocHost((void**)&HostResult, Length) );
```

```
  memset(HostResult, 0, Length);
  CUDA_SAFE_CALL( cudaMalloc((void**)&InData1, sizeof(float) * Length)
      );
  CUDA_SAFE_CALL( cudaMalloc((void**)&InData2, sizeof(float) * Length)
      );
  CUDA_SAFE_CALL( cudaMalloc((void**)&Result, sizeof(float) * Length) )
      ;

  /* ------- SET INPUT VALUEs ------- */
  InitData1 = (float*) malloc(sizeof(float) * Length);
  InitData2 = (float*) malloc(sizeof(float) * Length);
  for (int i = 0; i < Length; ++i) {
    InitData1[j] = (float) rand(); InitData2[j] = (float) rand();
  }
  CUDA_SAFE_CALL( cudaMemcpy( InData1, InitData1, sizeof(float) *
      Length, cudaMemcpyHostToDevice));
  CUDA_SAFE_CALL( cudaMemcpy( InData2, InitData2, sizeof(float) *
      Length, cudaMemcpyHostToDevice));

  /* ------- RUN COMPUTE KERNEL ------- */
  int n = 16 * 1024 * 1024;
  dim3 threads = dim3(512, 1);
  dim3 blocks  = dim3(n / threads.x, 1);
  saxpyCUDAKernel<<< blocks, threads, 0, stream>>>( a, InData1, InData2
      , OutData );

  /* ------- GET RESULT ------- */
  cudaMemcpy(HostResult, Result , sizeof(float) * Length,
      cudaMemcpyDeviceToHost );

  /* ------- CLEAN UP and EXIT ------- */
  CUDA_SAFE_CALL(cudaFree(InData1)); CUDA_SAFE_CALL(cudaFree(InData2));
  CUDA_SAFE_CALL(cudaFree(Result));  CUDA_SAFE_CALL(cudaFree(HostResult
      ));
  free(InitData1); free(InitData2);
  return 0;
}
```

Listing 3.2: CUDA version of the execution setup code for *saxpyCUDAKernel*.

CUDA gives fine control on non-functional aspects of stream computing, such as device management, resource management, code generation, *Kernel* loading and execution. At the same level of abstraction it is possible to implement a *Kernel* code. This is one of the main difference between CUDA and CAL/IL [20, 85]. In the latter, we'll show how the *Kernel* must be implemented using an assembly-like language.

Nvidia PTX. It defines a virtual machine and ISA for general purpose parallel thread execution [84]. PTX programs are translated at install time to the target hardware instruction set. The PTX-to-GPU translator and driver enable Nvidia

GPUs to be used as programmable parallel computers. High level language compilers for languages such as CUDA and C/C++ generate PTX instructions, which are optimized for and translated to native target-architecture instructions. The goals for PTX include the following:

- Provide a stable ISA that spans multiple GPU generations.
- Achieve performance in compiled applications comparable to native GPU performance.j
- Provide a machine-independent ISA for C/C++ and other compilers to target.
- Provide a code distribution ISA for application and middleware developers.
- Provide a common source-level ISA for optimizing code generators and translators, which map PTX to specific target machines.
- Facilitate hand-coding of libraries, performance *Kernels*, and architecture tests.
- Provide a scalable programming model that spans GPU sizes from a single unit to many parallel units.

While the specific resources available in a given target GPU will vary, the kinds of resources will be common across platforms, and these resources are abstracted in PTX through state spaces and data types. A state space is a storage area with particular characteristics. All variables reside in some state space. A variable declaration describes both the variables type and its state space.

The characteristics of a state space include its size, addressability, access speed, access rights, and level of sharing between threads:

- `Registers` (*.reg* state space) are fast storage locations. The number of registers is limited, and will vary from platform to platform. When the limit is exceeded, register variables will be spilled to memory, causing changes in performance. Registers may be typed (signed integer, unsigned integer, floating point, predicate) or untyped.
- The `global` (*.global*) state space is memory that is accessible by all threads in a context. It is the mechanism by which different CTAs and different grids can communicate. Use *ld.global*, *st.global*, and *atom.global* to access global variables. For any thread in a context, all addresses are in global memory are shared. Global memory is not sequentially consistent, thus there can be race conditions between threads. Therefore, developers must adopt lock-free and wait-free style programming.
- The `local` state space (*.local*) is private memory for each thread to keep its own data. It is typically standard memory with cache. The size is limited, as it must be allocated on a per-thread basis. Use *ld.local* and *st.local* to access local variables.

- The `parameter` state space is used for many reasons. One of these is to pass input arguments from the host to the *Kernel*. Each *Kernel* function definition includes an optional list of parameters. These parameters are addressable, read-only variables declared in the *.param* state space. The location of parameter space is implementation specific. Values passed from the host to the *Kernel* are accessed through these parameter variables using *ld.param* instructions. The *Kernel* parameter variables are shared across all CTAs within a grid. The address of a *Kernel* parameter may be moved into a register using the *mov* instruction. The resulting address is in the *.param* state space and is accessed using *ld.param* instructions.
- The `shared` (*.shared*) state space is a per-CTA region of memory for threads in a CTA to share data. An address in shared memory can be read and written by any thread in a CTA. Use *ld.shared* and *st.shared* to access shared variables.

In listing 3.3 we present a PTX equivalent implementation of listing 3.1.

```
.entry saxpyPTXKernel (
    .param .f32 a,
    .param .u64 InData1,
    .param .u64 InData2,
    .param .u64 Result)
  {
  .reg .u16 %rh<4>;
  .reg .u32 %r<5>;
  .reg .u64 %rd<10>;
  .reg .f32 %f<6>;
  cvt.u32.u16   %r1, %tid.x;
  mov.u16   %rh1, %ctaid.x;
  mov.u16   %rh2, %ntid.x;
  mul.wide.u16  %r2, %rh1, %rh2;
  add.u32   %r3, %r1, %r2;
  cvt.u64.s32   %rd1, %r3;
  mul.lo.u64  %rd2, %rd1, 4;
  ld.param.u64  %rd3, [InData2];
  add.u64   %rd4, %rd3, %rd2;
  ld.global.f32   %f1, [%rd4+0];
  ld.param.u64  %rd5, [InData1];
  add.u64   %rd6, %rd5, %rd2;
  ld.global.f32   %f2, [%rd6+0];
  ld.param.f32  %f3, [a];
  mad.f32   %f4, %f2, %f3, %f1;
  ld.param.u64  %rd7, [Result];
  add.u64   %rd8, %rd7, %rd2;
  st.global.f32   [%rd8+0], %f4;
  exit;
}
```

Listing 3.3: PTX version of the *saxpy* function.

IL gives fine control on *Kernel* implementations, due to the assembly level language provided. However, this low level programming language makes it very difficult to start write stream computations, keeping the learning curve very high.

AMD Brook+. AMD Brook+ [21] is an extension to the C language for cross-platform stream programming as well as Nvidia CUDA; it provides syntax to represent and manipulate streams, to specify *Kernels*. It is an open source standard designed to be simple to use, able to interoperate with multiple back-end, such as DirectX and OpenGL.
The Brook+ program structure can be modeled around a graph where nodes manipulate data and arcs indicating the flow of data through the system. A node can either restructure data or perform computations, but not both. Nodes that restructure data are referred to as stream operators, while nodes that perform computations as *Kernels*. An arc (stream) connects two nodes. It does not provide any storage, instead maps the output of one node to the input(s) of one or more other nodes. Streams reside in GPU memory and their dimension is limited by GPU hardware restrictions. They cannot be accessed directly by the host application program, but stream operators must be used. All Brook+ operations are non-blocking unless synchronization is needed. Synchronization is handled by the Brook+ run-time. Hence, it is important to carefully evaluate the function calls order to have the required interleaving.
During *Kernel* implementation, a developer must consider some restrictions due to the *Kernel* nature: dynamic memory allocation is not allowed, and in general pointers are not supported, and neither recursion nor calls to non-*Kernel* functions from within a *Kernel* are allowed.

In listing 3.4 we present a Brook+ equivalent implementation of listing 3.1.

```
kernel void saxpyBrookKernel (double a, double x<>, double y<>, out
    double result<>) {
 result = a * x + y;
}
```

Listing 3.4: Brook+ version of the *saxpy* function.

Vectors are denoted by using “<>” squares and the “*out*” keyword indicated which parameter is the output stream of a given function: *result* in the example.
In order to setup and execute the *saxpyBrookKernel* function, the code in listing 3.5 is required. This has the same goal of listing 3.2 under Nvidia.

```
int main(int argc, char** argv) {
  unsigned int Length = 100;
  double a = 10.0;
  double* InData1; double* InData2;
```

```
  double* Result;

  /* ———— MEMORY ALLOCATION ———— */
  InData1 = (double *) malloc ( sizeof(double) * Length);
  InData2 = (double *) malloc ( sizeof(double) * Length);
  Result = (double *) malloc ( sizeof(double) * Length);

  /* ———— SET INPUT VALUEs ———— */
  for (int i = 0; i < Length; ++i) {
    InData1[i] = (double) rand();
    InData2[i] = (double) rand();
  }

  /* ———— SET DOMAIN ———— */
  double indata1<Length>;
  double indata2<Length>;
  double result<Length>;
  streamRead ( indata1, InData1 );
  streamRead ( indata2, InData2 );

  /* ———— RUN COMPUTE KERNEL ———— */
  saxpyBrookKernel ( a, InData1, InData2, Result );

  /* ———— GET RESULT ———— */
  streamWrite (Result, result);

  /* ———— CLEAN UP and EXIT ———— */
  free(InData1);
  free(InData2);
  free(Result);
  return 0;
}
```

Listing 3.5: Brook+ version of the execution setup code for *saxpyBrookKernel*.

Brook+ gives an abstract view of GPU computation, so that developers can focus only on *Kernels* implementation, without to consider any GPU details. This high level programming language makes it easier to start write stream computations, keeping the learning curve low. However, all optimizations are entrusted to underlying framework.

AMD CAL. CAL is a device-driver layer that sits on top of AMD's Close-To-Metal Hardware Abstraction Layer. CAL provides features like devices and resources management, code generation, and *Kernel* loading and execution. The CAL programming interface consists of:

- A minimal set of extensions to the C language that allow the programmer to target portions of the source code for execution on the device.
- A run-time library split into:

– Host application, that performs application work, sends commands to the GPU using CAL API.
– GPU *Kernel*, that reads input data, performs stream parallel computation and writes output data.
– Common component, that provides built-in vector types and a subset of the C standard library that are supported in both host and device code.

The computational model is processor independent and allows the user to switch easily between directing a computation from GPU to CPU and vice versa. The API should permit a dynamic load balancer to be written on top of CAL. A CAL system comprises of one master process executing on the CPU and driving one or more devices. A device is a hardware component capable of running CAL programs (*Kernels*). A device has one or more computational processors. The *Kernel* is executed on these processors and is implemented by using the AMD Intermediate Language (IL). A device is connected to two memory sub-systems: local and remote. The master process can read and write to both local and remote memory of any device, though typically, the master has higher read and write speeds to the remote memory of a device. The master process submits commands for execution using a device context. The master process is also capable of querying the context for the status of the completion of these tasks. The inputs and outputs to the program can be set up to reside either in local or remote memory. A computation is invoked by setting up one or more outputs and specifying a region (domain of execution) into this output that must be computed. In the case of a device having multiple processors (such as a GPU device), a scheduler distributes the workload region to various SIMD processors on the device. The CAL abstraction divides commands into two key types: device commands and context commands. The device commands primarily involve resource allocation (local or remote memory). A context is a queue of commands that are sent to a device. It is possible to have parallel queues for different parts of the device. Resources are created on devices and are mapped into contexts; this must be done to provide scoping and access control from within a command queue. Each context represents a unique queue. Each queue operates independently of each other. The context commands (e.g. *calCtxRunProgram*) queue their actions in the supplied context. The device does not execute the commands until the queue is flushed; this occurs implicitly when the queue is full or explicitly through CAL API calls. Data sharing across contexts is possible by mapping the same resource into multiple contexts. Synchronization of multiple contexts is the client's responsibility.
In listing 3.6 we present a CAL equivalent implementation of listing 3.5. This is required to setup and execute the *Kernel* function implemented in listing 3.7.

```
int main( int argc, char** argv ) {
  /* ———— INITIALIZATION ———— */
  calInit();
```

```
CALuint numDevices = 0;
calDeviceGetCount( &numDevices );
CALdeviceinfo info;
calDeviceGetInfo( &info, 0 );
CALdevice device = 0;
calDeviceOpen( &device, 0 );
CALcontext ctx = 0;
calCtxCreate( &ctx, device );

/* ———— COMPILE & LINK KERNEL ———— */
CALdeviceattribs attribs;
attribs.struct_size = sizeof(CALdeviceattribs);
calDeviceGetAttribs(&attribs, 0);
CALobject obj;
calclCompile(&obj, CAL_LANGUAGE_IL, saxpyILKernel, attribs.target);
// Link object into an image
CALimage image = NULL;
calclLink(&image, &obj, 1) ;

/* ———— MEMORY ALLOCATION ———— */
// allocate input/output resources and map them into the context
unsigned int Length = 100;
CALresource InData1 = 0; CALresource InData2 = 0;
calResAllocLocal1D(&InData1, device, Length, CAL_FORMAT_DOUBLE_1, 0)
    ;
calResAllocLocal1D(&InData2, device, Length, CAL_FORMAT_DOUBLE_1, 0)
    ;
CALresource Result = 0;
calResAllocLocal1D(&Result, device, Length, CAL_FORMAT_DOUBLE_1, 0);
CALresource a = 0;
calResAllocRemote1D(&a, &device, 1, 1, CAL_FORMAT_DOUBLE_1, 0);
CALuint pitch1 = 0, pitch2 = 0;
CALmem InMem1 = 0, InMem2 = 0;

/* ———— SET INPUT VALUEs ———— */
double* findata1 = NULL; double* findata2 = NULL;
calCtxGetMem(&InMem1, ctx, InData1);
calCtxGetMem(&InMem2, ctx, InData2);
calResMap((CALvoid**)&findata1, &pitch1, InData1, 0);
for (int i = 0; i < Length; ++i) findata1[i * pitch1] = rand();
calResUnmap(InData1);
calResMap((CALvoid**)&findata2, &pitch2, InData2, 0);
for (int i = 0; i < Length; ++i) findata2[i * pitch2] = rand();
calResUnmap(InData2);
double* constPtr = NULL;
CALuint constPitch = 0; CALmem constMem = 0;

// Map constant resource to CPU and initialize values
calCtxGetMem(&constMem, ctx, a );
calResMap((CALvoid**)&constPtr, &constPitch, a, 0);
constPtr[0] = 10.0;
```

```
  calResUnmap( a );

  // Mapping output resource to CPU and initializing values
  void* res_data = NULL;
  CALuint pitch3 = 0;
  CALmem OutMem = 0;
  calCtxGetMem(&OutMem, ctx, Result);
  calResMap(&res_data, &pitch3, Result, 0);
  memset(res_data, 0, pitch3 * Length * sizeof(double));
  calResUnmap(Result);

  /* ———— LOAD MODULE & SET DOMAIN ———— */
  CALmodule module;
  calModuleLoad(&module, ctx, image);
  CALfunc func;
  CALname InName1, InName2, OutName, ConstName;
  calModuleGetEntry( &func, ctx, module, "main" );
  calModuleGetName(&InName1, ctx, module, "i0");
  calModuleGetName(&InName2, ctx, module, "i1");
  calModuleGetName(&OutName, ctx, module, "o0");
  calModuleGetName(&ConstName, ctx, module, "cb0");
  calCtxSetMem(ctx, InName1, InMem1);
  calCtxSetMem(ctx, InName2, InMem2);
  calCtxSetMem(ctx, OutName, OutMem);
  calCtxSetMem(ctx, ConstName, constMem);
  CALdomain domain = {0, 0, Length, 1};

  /* ———— RUN COMPUTE KERNEL ———— */
  CALevent event;
  calCtxRunProgram( &event, ctx, func, &domain );

  // wait for function to finish
  while (calCtxIsEventDone(ctx, event) == CAL_RESULT_PENDING) { };

  /* ———— GET RESULT ———— */
  CALuint pitch4 = 0;
  double* foutdata = 0;
  calResMap((CALvoid**)& foutdata, & pitch4, Result, 0);
  for (int i = 0; i < Length; ++i)
  foutdata [i * pitch4] = (double)(i * pitch4);
  calResUnmap(Result);
}
```

Listing 3.6: CAL version of the execution setup code for *saxpyILKernel*.

CAL gives finer control on non-functional aspects of stream computing, such as device management, resource management, code generation, *Kernel* loading and execution.

AMD IL. It is a pseudo-assembly language that can be used to develop both graphics programs (vertex, geometry and pixel shaders) and general purpose data-

parallel programs (*Kernels*). IL is designed so that programs can be developed to run on a variety of platforms without having to be rewritten for each one. IL is a type-less language, thus variables, parameters, etc., have no specific type and can represent 32-bit signed integers, unsigned integers and floating point numbers, and 64-bit double precision floating point numbers (only supported when the underlying platform supports double precision numbers) without first defining the variable type. Number types are defined by the instruction. IL use makes optimizing the *Kernels* easier, since programmers have more control over memory allocation, register and instruction set used without requiring deep knowledge of the GPU architecture. At the same time, similarly to all assembly languages, the learning curve is higher than high-level languages such as C. Providing the Brook+ language for *Kernels*, AMD enables non-HPC programmers to take advantage from stream processor performance.
In listing 3.7 we present a IL equivalent implementation of listing 3.4.

```
const CALchar* saxpyILKernel =
"il_ps_2_0\n"
"dcl_input_position_interp(linear_noperspective) v0.xy__\n"
"dcl_output_generic o0.x___\n"
"dcl_cb cb0[1]\n"
"dcl_resource_id(0)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)
    _fmtw(float)\n"
"dcl_resource_id(1)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)
    _fmtw(float)\n"
"sample_resource(0)_sampler(0) r0, v0.xyxx\n"
"sample_resource(1)_sampler(0) r1, v0.xyxx\n"
"mad_ieee o0.x___, cb0[0].x, r0.x, r1.x\n"
"ret_dyn\n"
"end\n";
```

Listing 3.7: IL version of the *saxpy* function.

IL gives finer control on *Kernel* implementations than Brook+, due to the assembly level language provided. However, this low level programming language makes it very difficult to start write stream computations, keeping the learning curve very high.

The GPGPU compilation toolchain

The fragmentation in SIMD architectures and languages creates an interoperability problem that hinders the wide adoption of stream computing. To run any stream program on any stream architecture, one must develop a separate compiler for every language and architecture pair. For these reasons, each GPUs vendor adopts a two-level compilation approach that can mitigate the engineering complexity of developing a new stream language: a High-Level Compiler (HLC) and a Low-Level Compiler (LLC). The HLC accepts source code written in a stream language (e.g.

AMD Brook+, AMD CAL and Nvidia CUDA) and the description of an architecture via an abstract machine model, and it produces a mapping of logical stream constructs to the physical resources available in the given architecture. Currently, the mapping is expressed using APIs which are built upon the C programming language. The high-level languages, AMD CAL and Nvidia CUDA, hide enough details about the underlying structure of the system so that the programmer can program GPUs without knowledge of the underlying system.

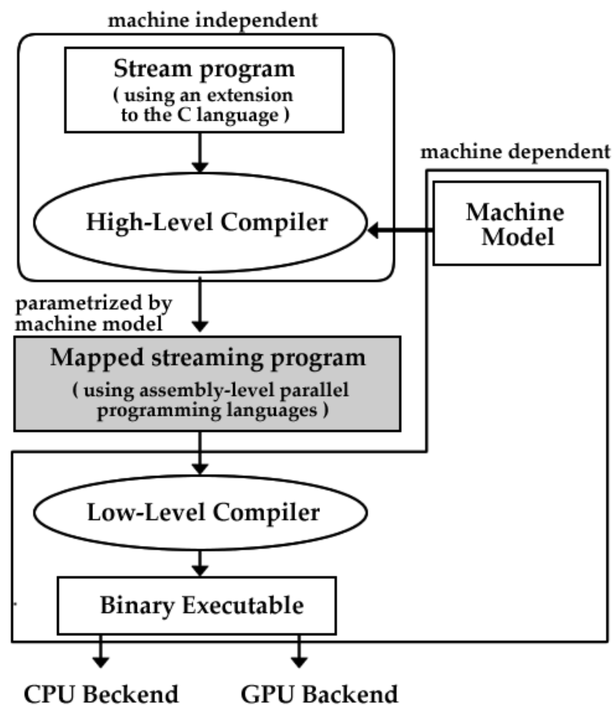

Figure 3.1: Two-level compilation approach.

Both vendors defined a low-level virtual machine model, in order to expose the GPU as a data-parallel computing device: AMD CAL is based on the Close-to-the--Metal Hardware Abstraction Layer [106], whereas Nvidia CUDA is based on the Parallel Thread eXecution (PTX).

The LLC produces architecture-specific assembly language. The assembly-level parallel programming languages, AMD IL and Nvidia PTX, provide capabilities for fine-grained synchronization and data sharing across hardware threads, so that programmers have more control over *Kernels* optimization. Both are a type less language based on virtual register architecture.

OpenCL. Recently, Khronos Group, a consortium of very important companies and institutions, has created a standard for cross-platform, parallel programming of modern processors, such as GPGPUs and IBM Cell, called Open Computing Language (OpenCL) [107]. It is an open industry standard for programming a heterogeneous collection of CPUs, GPUs and other discrete computing devices organized into a single platform. OpenCL is a framework for parallel programming and includes a language, API, libraries and a run-time system to support software development. Using OpenCL, for example, a programmer can write general purpose programs that execute on GPUs without the need to map their algorithms onto a 3D graphics API such as OpenGL or DirectX. OpenCL provides a low-level hardware abstraction and a framework to support programming and many details of the underlying hardware are exposed.

Its platform model consists of a host connected to one or more OpenCL devices. An OpenCL device is divided into one or more compute units (CUs) which are further divided into one or more processing elements (PEs). Computations on a device occur within the processing elements. An OpenCL application runs on a host according to the models native to the host platform. The OpenCL application submits commands from the host to execute computations on the processing elements within a device. The processing elements within a CU execute a single stream of instructions as SIMD units (execute in lockstep with a single stream of instructions) or as SIMD units (each PE maintains its own program counter).

The execution of an OpenCL program occurs in two parts: *Kernels* that execute on one or more OpenCL devices and a host program that executes on the host. The host program defines the context for the *Kernels* and manages their execution. The core of the OpenCL execution model is defined by how the *Kernel* execute. When a *Kernel* is submitted for execution by the host, an index space is defined. An instance of the *Kernel* executes for each point in this index space. This *Kernel* instance is called a work-item and is identified by its point in the index space, which provides a global ID for the work-item. Each work-item executes the same code but the specific execution pathway through the code and the data operated upon can vary per work-item. Work-items are organized into work-groups which provide a more coarse-grained decomposition of the index space. They are assigned a unique work-group ID with the same dimensionality as the index space used for the work-items. Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination

of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the processing elements of a single CU.

The execution context includes the following resources:

- Devices: The collection of OpenCL devices to be used by the host.
- *Kernels*: The OpenCL functions that run on OpenCL devices.
- Program Objects: The program source and executable that implement the *Kernels*.
- Memory Objects: A set of memory objects visible to the host and the OpenCL devices. Memory objects contain values that can be operated on by instances of a *Kernel*.

The context is created and manipulated by the host using functions from the OpenCL API. The host creates a data structure called a command-queue to coordinate execution of the *Kernels* on the devices. The host places commands into the command-queue which are then scheduled onto the devices within the context. These include: *Kernel* execution commands (i.e. execute a *Kernel* on a device), memory commands (i.e. transfer data to, from, or between memory objects), and synchronization commands.

Work-item(s) executing a *Kernel* have access to four distinct memory regions:

- Global Memory. This memory region permits read/write access to all work-items in all work-groups. Work-items can read from or write to any element of a memory object. Reads and writes to global memory may be cached depending on the capabilities of the device.
- Constant Memory: A region of global memory that remains constant during the execution of a *Kernel*. The host allocates and initializes memory objects placed into constant memory.
- Local Memory: A memory region local to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group. It may be implemented as dedicated regions of memory on the OpenCL device. Alternatively, it may be mapped onto sections of the global memory.
- Private Memory: A region of memory private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item.

The application running on the host uses the OpenCL API to create memory objects in global memory, and to enqueue memory commands that operate on these memory objects.

The OpenCL execution model supports data parallel and task parallel programming models, as well as supporting hybrids of these two models. The OpenCL framework allows applications to use a host and one or more OpenCL devices as a single heterogeneous parallel computer system. The framework contains the following components:

- OpenCL Platform layer: The platform layer allows the host program to discover OpenCL devices and their capabilities and to create contexts.
- OpenCL Runtime: The run-time allows the host program to manipulate contexts once they have been created.
- OpenCL Compiler: The OpenCL compiler creates program executables that contain OpenCL *Kernels*. The OpenCL C programming language implemented by the compiler supports a subset of the ISO C99 language with extensions for parallelism.

Chapter 4

4-Centauri: a MSIL to NVIDIA PTX compiler

The high-level GPU programming languages (recall Section 3.2.4) provided by both Nvidia and AMD present those problems listed in Section 3.1. Indeed, the correct and efficient design of GPUs parallel programs requires the consideration of several different concerns that are difficult to separate during program development. How the computation is performed using processing elements (such as processes and threads), and how these communicate, are non-functional aspects of a program since they do not contribute to define the result of the computation but only how it is performed. Subscribing the separation of concerns concept, typical of Aspect Oriented Programming (AOP) [45], we recognize the importance of using proper tools to program the non-functional aspects related to parallelism exploitation. We propose that programmers use meta-data to "suggest" consciously how a parallel application can be automatically derived from the code. We will describe a proper meta-program, called **4-Centauri**, that can efficiently handle such meta-data. This is different from the standard AOP approach where join points are defined using patterns, making the programmer unaware of program transformation details that will be applied afterwards. By splitting the responsibility between the meta-program/compiler and the programmer we exploit the strengths of them, and avoid their weaknesses. Our underlying assumption is that the programmer can make better granularity and partitioning decisions, while the compiler can correctly manage synchronization. This simplifies the task of writing parallel programs, making the power of parallel and distributed systems more accessible.
In next section we describe the **4-Centauri**'s design and implementation.

4.1 Compiling from MSIL to PTX

The **4-Centauri** compiler translates a given program P_{CLR} (as defined in Section 2.5.1) into an equivalent program P_{PTX} (as defined in Section 2.5.7). In particular,

4-Centauri leverages the flexibility and cost-effectiveness of the CLR to translate MSIL codes, enriched with meta-data, into a modern mass-market special-purpose architecture intermediate language code, the Nvidia PTX.

Of course it is possible to implement different JIT compilers that can target special-purpose architectures. The problem is that CLR (and JVM) lacks the ability to express special-purpose architecture features in a form that is recognizable by the JIT module. On a standard CLR those features are hidden from the abstraction layer provided by the CIL.

In our approach **4-Centauri** uses standard mechanisms provided by CLR to represent different parallel computations and abstract away tasks of specifying parallelism, communication, synchronization, etc. Moreover, we want to preserve the illusion of the system directly executing the program as the programmer wrote it, with no user-visible optimizations.
To achieve our goals, our technique embodies the following design features:

- **Separation of control and data-intensive code**. *Kernel*s are considered a unit of computation expressed with languages targeting CLR. They operate on a set of input and output streams, may contain local state, and generally encapsulate data-parallel code. *Kernel*s can be asynchronously monitored and controlled from a control thread running on CPU processor.

- **Explicit communication via stream**. The streaming nature of the computation is exposed via stream abstraction. A stream is defined as an ordered collection of values of homogeneous type through which CPUs and GPUs can communicate.

- **Implicit memory management for streaming data**. The orchestration of data movement via Direct Memory Access (`DMA`) as well as the GPU memory hierarchy is hidden under the hood.

We have divided compilation, optimization, and specialization processes into different stages by introducing a programming toolchain that controls the generation process, as shown in Figure 4.1.

Along with **4-Centauri** a domain-specific library is provided with a set of types (e.g. *HPRAMLib* and *LogPLib* in Figure 4.1) that express different models of parallel execution. General application programmers can select which type to use in function of which model of parallel execution best fit in their programs. At deploy time, for each type selected if there is an underlying architecture that exposes the same model of execution, **4-Centauri** will translate that type's methods into that architecture executable code. If there is no special architectures available, those methods will execute on CPU. In order to map models, leaving the user to code and debug at source level, **4-Centauri** operates at the intermediate level, such that all its manipulation are transparent to developers.

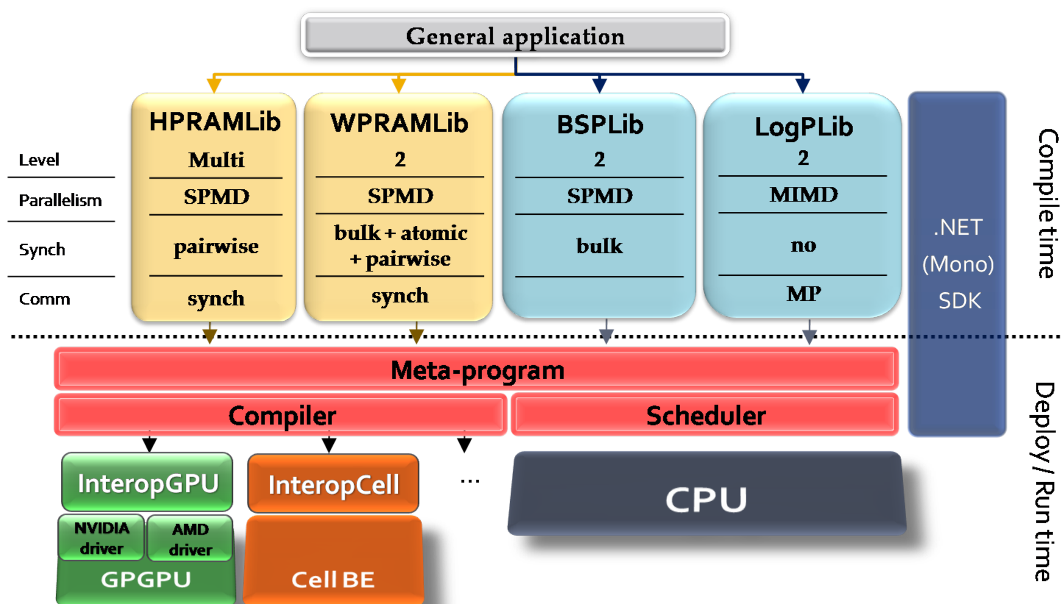

Figure 4.1: **4-Centauri** software stack.

4.1.1 The philosophy

The **4-Centauri** philosophy on parallel computing is the same of Mentat (recall Section 3.2.3) since it is guided by two observations:

1. the programmer has a better comprehension of the problem domain of the application and can make better data and computation partitioning decisions than compilers can.

2. the design and implementation of non-functional aspects of a parallel program, such as the management of tens to thousands of asynchronous tasks, where timing dependent errors are easy to make, is beyond the capacity of most programmers unless a tremendous amount of effort is expended. Compilers, on the other hand, are very good at ensuring that events happen in the right order, and can more readily and correctly manage communication and synchronization, particularly in highly asynchronous, non-SPMD, environments.

In essence, we design our framework such that it supports users at three different levels of abstraction. At the *higher* level, general application programmers have no access to any information about execution environment, they just make a reference to a domain-specific library (DSLib) (see Figure 4.1) in the usual way, as for Mentat, MapReduce, RapidMind. Eventually, they can take advantage of the computational power of the GPUs. At this level we want to preserve the value of legacy codes, as for ADAPTOR, SUIF and pMapper compilers, without requiring

programmers must learn another language and must re-implement their existing code, as for RapidMind, and OpenMP. Moreover, programmers can continue to debug their programs in their sequential form, as for ADAPTOR, SUIF, pMapper, and Mentat.

At the *middle* level, experts in a particular (computational) domain, but not necessarily in computer graphics, can develop libraries (i.e. DSLib) leveraging the stream programming model provided by our framework. At this level, architecture details are transparent to programmers, they just use the streaming capabilities of the underlying GPU processors exposed by a dedicated API. This is the same approach provided by X10 and Muskel on multi-core architectures.

At the *lower* level, experts in computer (graphics) architectures and programming models can develop their own meta-program in order to map high-level code to new architectures.

4.1.2 Two-level compilation model

4-Centauri meta-program is based on the two-level compilation model, as shown in Figure 4.2, as for Nvidia CUDA and AMD CAL.

The high-level compiler is one of those targeting the CLR, since the idea is to make architecture primitives available to the CLR-based program without changing the design of the CLR itself. The low-level compilation is provided by the LLCs (introduced in Section 3.2.4) by AMD and Nvidia. **4-Centauri** is the interface between the two levels. Therefore, **4-Centauri** is composed by several modules, three in Figure 4.1. The *Meta-program* exposes the API describes in the following section, and used in the DSLibs implementation, i.e. defined at middle level of abstraction. The *Scheduler* manages execution of CPU- and GPU-part of code, mapping the execution domain expressed in the source code by DSLibs to available underlying processing units (e.g. GPU and CPU). The *Compiler* translates the part of MSIL code executed by an "accelerator" (e.g. GPU), and passed as input by the *Scheduler*, into low-level language code of the underlying architecture. In order to execute that part of code on a new architecture, a new *Compiler* that targeting that architecture must be added to the **4-Centauri** framework.

Application Program Interface

Considering the GPGPU computation model introduced in Section 1.4, we define a streaming computational model that explicitly provides the dependency between all stream operations to each execution engine. Therefore, programmers are aware of the additional operation costs due to engines communication via streams. To

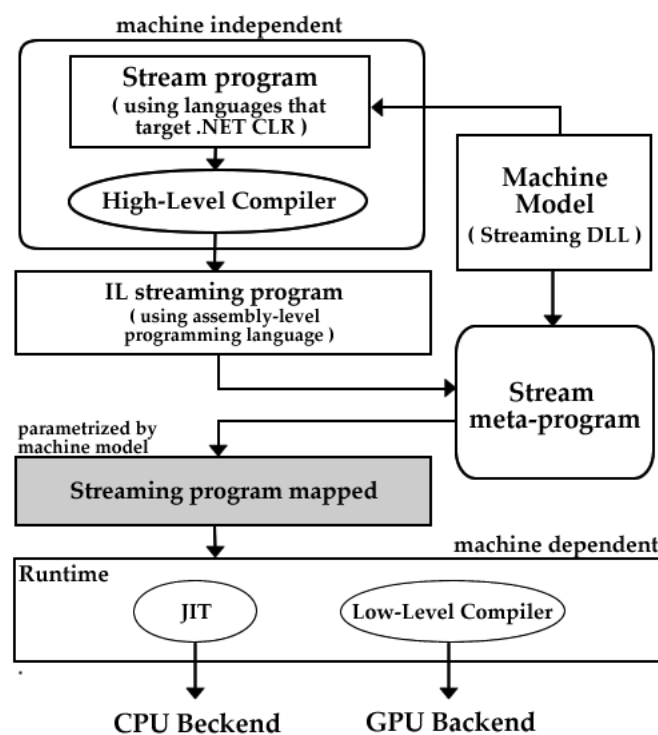

Figure 4.2: Two-level compilation model.

encapsulate the streaming application as transformed by a high-level compiler we define the following logical entities:

- *Stream* objects are used to assign data to specific hardware locations in the stream processors local memories and to refer to locations in the global memory (RAM) for DMA transfers. They contain an ordered collection of data elements of a given type. From the point of view of the controlling program, streams are accessible *sequentially* and implemented as a circular buffer assigned to a location in a memory. As shown in Listing 4.1, the Stream objects are implemented by two generic types: `InputStream<T>` and `OutputStream<T>`,

that inherit from `Data<T>` introduced in Listing 2.9. They change how data elements are accessed. The former allows to read the next element from an input stream, and the latter allows to write one element into an output stream. Since *Kernel* entities communicate with Control entities through Stream entities then *kernel*-annotated methods must have those types as parameters.

```
public class InputStream<T> : Data<T> {
    public override int Count() {...}
    public override void Reset() {...}
    public override bool MoveNext() {...}
    public T Current {...}

    /* For workload distribution purpose */
    public List<InputStream<T>> Split(int num_array) {...}
}

public class OutputStream<T> : Data<T> {
    public override int Count() {...}
    public override void Reset() {...}
    public override bool MoveNext() {...}
    public T Current {...}

    /* For workload scattering and gathering purposes */
    public List<OutputStream<T>> Split(int num_array) {...}
    public void Concat(OutputStream<T>[] nstream)
}
```

Listing 4.1: C# implementation of GPGPU streams.

In Listing 4.2 we will provide an example of these stream types in use.

- *Kernel* objects are used to map methods to stream processors. Programmers can declare which methods in a data type must execute on GPGPUs. Their code consume zero or more input streams and produce zero or more output streams. In order to use standard mechanism provided by CLR, we have decided to use a Custom Attributes (CA) type, rather than to introduce a new data type for *kernel*s. Doing this, our code is cross-platform, since CA are ignored by the execution engine, but provided along with binary files.

```
public sealed class Kernel : Attribute {
    //...
}
```

By this attribute, developers annotate which methods should execute on a GPGPU, and which implement control code.

- *Controls* implement control code. They can initiate, monitor, and terminate the execution of *kernel*s. These are transparent to programmers.

Actually, **4-Centauri** provides only one type that represents the GPGPU model of execution, and a MSIL to Nvidia PTX compiler. However, we are working on new types for other models, such as LogP and H-PRAM, and it is almost ready the AMD IL Code Generator module of **4-Centauri**.

Example of implementation: data-parallel on GPGPU

In order to give an idea of a possible implementation of formal concepts expressed in previous sections, now we provide the same example of Section 2.5.4, i.e. data-parallel operations, but this time considering the GPGPU model of execution.

In the following listing 4.2 the *GPUDataParallel* class implements both *Map* and *Reduce* operations that take stream data types as arguments for scattering input data to and gathering results from a GPU.

```
/* Available operations for reduction purposes */
public enum DPOper {
  Add = 1,
  Prod = 2
}

/* Example of delegate definition */
public delegate void DPSTREAM<T,K>(OutputStream<T>[] outputs,
   InputStream<K>[] inputs);

/* Implements data-parallel operations for GPGPUs architectures */
public class GPUDataParallel<T,K> : DataParallel<T,K>
{
  /* GPU configuration and execution management related fields */
  //...
  GPUFunction fun1;
  GPUContext ctx1;
  GPUModule mod1;

  /* VEE-side memory managements */
  int[] ptrInput;
  GCHandle[] gcHdInput;

  public GPUDataParallel(Object ob) : base(ob)
  {
    /*  Initialization of: */
    1. CUDADriver environment
    2. Compiler data structure for code analysis.
  }

  /* CUDADriver environment exit */
  ~GPUDataParallel() {
      CUDA.Exit(job1);
  }

  public virtual void Map( string method_name,
```

```
                                 OutputStream<T>[] outputs,
                                 InputStream<K>[] inputs ) {

        /* List of calls performed by the 4-Centauri on a Kernel having '
           method_name' as name. */

           1. Set up CUDADriver environment, context, load binary file
              and the \PTX function
              resulting from compilation of method m

           2. Scatter data from CPU to GPU.
              For each input stream
              2.1. Allocate GPU memory
              2.2. Copy data values from current input stream to GPU
                   memory

           3. Allocate GPU memory for the output stream

           4. Set up CUDADriver specific parameters

           5. Launch GPU execution

           6. Gather data from GPU to CPU
    }

    public virtual void Reduce( string method_name,
                                DPoper operation,
                                OutputStream<T>[] outputs,
                                InputStream<K>[] inputs )
    {
        /*
           Map steps, having added the reduction operation to kernel code
        */
    }
}
```

Listing 4.2: C# implementation of data-parallel operations on GPGPU architecture.

In 4.2 we list *Map* steps that exploit **4-Centauri** API to execute a *Kernel* on a GPU. At step 1, the MSIL Kernel code is loaded and analyzed by **4-Centauri**. It scans its body to evaluate which memory resources are referred. Moreover, it reconstructs the stack behavior and translates MSIL code to Nvidia PTX code. At step 2 and 3, **4-Centauri** performs a set of calls required to allocate memory on a GPU and send data to it. At step 4-5, **4-Centauri** configures and launches the execution of PTX Kernel code. At the end, results are gathered from GPU and returned to *Map* caller via *outputs* parameter.

In Section 4.2 we will describe into details which modules of **4-Centauri** compiler perform those steps. At higher level of abstraction in their application programmers must:

1. implement a class having at least one *Kernel* annotated method.

2. instantiate *GPUDataParallel* class.

3. invoke one of the operation provided by *GPUDataParallel* class, passing the name of a Kernel defined at the point 1 as parameter.

An example of application is shown in the listing 4.3.

```
public class MyClass {
  //...

  [Kernel]
  public void Add(OutputStream<T> outputs, InputStream<float>[] inputs
      ) {
      outputs[0].Current = inputs[0].Current + inputs[1].Current;
  }
}

public class Program {
  //...

  static void Main(string[] args)
  {
      MyClass mc = new MyClass();
      int[] A;
      int[] B;
      int[] C;

      /* Vectors initialization */
      //...

      OutputStream<int>[] outputs = new OutputStream<int>[1];
      outputs[0] = new OutputStream<int>(C);

      InputStream<int>[] inputs = new InputStream<int>[2];
      inputs[0] = new InputStream<int>(A);
      inputs[1] = new InputStream<int>(B);

      GPUDataParalle<int,int> dp = new GPUDataParallel<int,int>(mc);
      dp.Map("Add", outputs, inputs);
}
```

Listing 4.3: C# application that exploit data-parallel operations defined in listing 4.2.

The code in listing 4.3 can be compiled with a standard C# compiler. The MSIL code obtained after the compilation is a standard one, thus it can be executed on all standard CLR or Mono VM. At this point the code is not mapped on a GPGPU yet. Another step is required: the **4-Centauri** compiler execution.

4.2 Organization of the compiler

As shown in Figure 4.3, the **4-Centauri** system consists of the following parts:

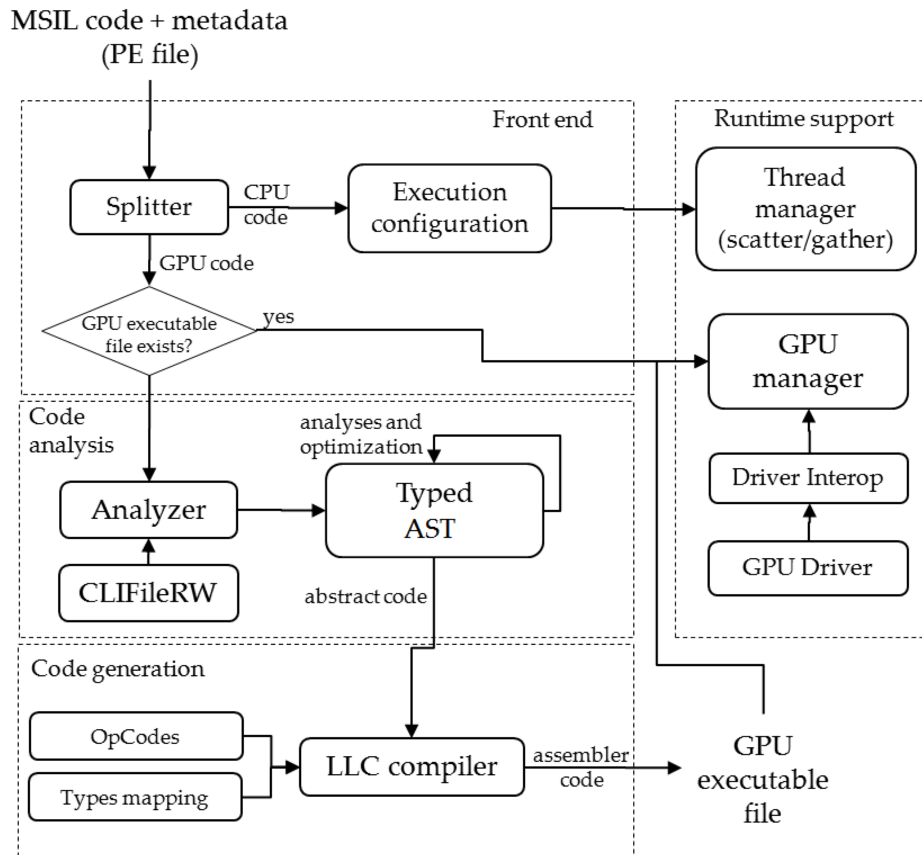

Figure 4.3: Organization of the **4-Centauri** compiler.

- *Analyzer*. It scans a given MSIL code looking for Kernel-annotated methods. If there is at least one GPU available, for each Kernel found, the *Analyzer* checks whether it is already compiled or not. In the former case, the *Analyzer* configures the execution environment (e.g. memory allocation and initialization, context and function definition, etc.) and invokes the run-time support to start *Kernel* execution. In the latter case, it scans *Kernel* MSIL code in order to evaluate which resources (e.g. shared and global memory, registers, etc.) are referred.

- *Parser*. It takes a MSIL code as input and builds an Abstract Syntax tree (*AST*) by using an abstract stack.

- *Code Generator*. It visits an AST and translates its nodes into a specific intermediate language depending on the available underlying GPU, such as

Nvidia PTX. Finally, it produces an executable file such that in the following executions of the same code, no compilation step is required.

- *Runtime support*. It provides support for executing CPU and GPU computations, such as scattering input data, and gathering results at the end of computation. If part of the computation is executed on a GPU, our support performs all that is necessary to configure and execute *Kernel* code.

From Section 4.2.2 through 4.2.5 each of the **4-Centauri** components is described in detail.

4.2.1 Data Structures

We think it is important first to introduce the data structures that have been defined and will be used, by **4-Centauri** for compilation purposes.

- ***StreamArgs***. It represents a stream with an associated type. As stated in PTX specification [84] “an input/output stream is addressable either by an integer or bit-size type register reg containing a byte address, or a sum of register reg containing a byte address plus a constant integer byte offset.” Therefore, in order to manage stream access, i.e. single access or in a loop over its elements, we define the following tuple of register identifiers:

  $$regIdentifier \equiv kReader \times kIncrem \times kIndexer \times kLoopBound \times kLoopIndex$$

  where

  - *kReader* register that holds the reference to the first element of the stream.
  - *kIncrem* register that holds the amount of the increment to address the next element of the stream.
  - *kIndexer* register that holds the iterator over the stream.
  - *kLoopBound* register that holds the stream length.
  - *kLoopIndex* register that holds the loop indexer.

  Actually, the last two registers are optional, since they depend on whether there is a loop over a stream or not.

- Abstract ***Instruction***, named AI. It is an abstract assembly-like instruction that is not tailored to a specific GPU. An instruction is modeled as follows:

```
class Instruction {
  private OpCode _op;
  private Predicate _pred;
  private string[] _operands;
  private Label _label;
```

```
  public Instruction(Predicate p, Label lb, OpCode o, string[]
      opers) {...}

  /* Return a complete instruction in the correct syntax */
  public override string ToString() {...}
}
```

where *Predicate* (e.g. *p*) is a guard that determines whether an instruction or instruction block can be executed or not. It is used in control flow instructions (e.g. *setp*) but also to avoid illegal execution, such as

```
setp.eq.f32 p,y,0;       // p = (y == 0)
@!p div.f32 ratio,x,y   // if y == 0 then ratio = x / y
```

OpCode is a generic operation code, *Label* is a branch target of an instruction, thus for control flow purposes, and *_operands* is a set of operands required by an instruction.

- ***Kernel***. It represents an abstract Kernel that is modeled as follows:

```
public class Kernel {
  /* Kernel unique name */
  private string _name;

  /* Input and Output streams used by a kernel */
  private Dictionary<int,StreamArgs> _streams;

  /* Set of registers used by a kernel */
  private Registers_registers;

  /* Shared memory variables used by a kernel */
  private List<SharedMem> _sharedmem;

  /* kernel intermediate code */
  private List<Instruction> _streamCode;

  private int _idRegs;
  private int _idPredicate;
  public Queue<int> idPredicate;

  public int AddRegID(OperandType opt) {...}
  public int AddPredicate() {...}
  public int GetPredicate() {...}
  public virtual void AddSharedMem(OperandType opt, int align,
      string name, int qta) {...}

  /* Emit the Arithmetic abstract instruction */
  public virtual void EmitAR(...) {...}

  /* Emit the Comparison abstract instruction */
```

```
  public virtual void EmitCMP(...) {...}

  /* Emit the Data movement abstract instruction */
  public virtual void EmitMV(...) {...}

  /* Emit the Memory Load abstract instruction */
  public virtual void EmitLS(...) {...}

  /* Emit the Conversion abstract instruction */
  public virtual void EmitCV(...) {...}

  /* Emit the Control Flow abstract instruction */
  public virtual void EmitFC(...) {...}

  /* Append an abstract instruction to the kernel intermediate
     code */
  private void InternalEmit(Instruction instr) {...}
}
```

Using the notation introduced in Section 2.5.1, a kernel $k \in \mathbb{FUN}_{GPGPU}$ refers to a set of memory resources $\in \mathbb{MEM}_{GPGPU}$, in particular a set of *_streams* $\in \mathbb{RG}_{GPGPU}$, a set of *_registers* $\in \mathbb{RG}_{GPGPU}$, a set of shared memory units $\in \mathbb{SM}_{GPGPU}$. The body of k is held in *_streamCode*, that is the result of applying η_{GPGPU} function to k.

We define methods to set/get memory resources associated with a Kernel, and a set of emitters for each class of AIs: arithmetical-logic, comparison, data movement, parameter access, type conversion and control flow.

- ***Node***. It is an abstract interpretation of a MSIL instruction (AbI). It is modeled by the following class

```
/* Base class */
class Node {
    protected int id;
    protected OperandInfo opcode;
    protected int regId;
    protected Label label;
    protected List<Node> nodes;

    public Node(Label lab, int identifier, OperandInfo opc) {...}

    public void AddNode(Node nd) {...}
    public virtual string Scan() {...}
}
```

The information required on a MSIL instruction for the generation step is summarized by the *Node* class fields: operation code (*opcode*), label (*label*), a list of children nodes (*nodes*) required to compute the current instruction, and the identifier of the register (*regId*) that will hold the result of the current

instruction execution.
The *Scan* method generates the AI relative to the current AbI. To accomplish this task, it invokes the Scan method of each child, if present, and stores its results into a register, that will be an operand of the current AbI. We define a set of classes that extend *Node* one in order to map each AbI to an AI. In table 4.1 this mapping is defined between *Node* classes and MSIL instructions sets.

MSIL instruction type	**Equivalent Node class**
stloc	NodeLSloc
ldloc	NodeConst
ldarg	NodeARG
ldc_Ix	NodeConst
conv_Ix	NodeCVT
call, calli, callvirt	NodeCall
br	NodeBRA
brtrue, brfalse	NodeCONDBRA
beq, bge, bgt, ble, blt	NodeFCBRA
ceq, cgt, clt	NodeSetCMP
add, mul, sub, div	NodeCALC

Table 4.1: Mapping of MSIL instructions and Node classes. For the sake of brevity we do not consider efficient encoding versions of MSIL instructions. For a complete list, please see the ECMA 335, partition III [1].

4.2.2 Code Analyzer

In Figure 4.3, the *Analyzer* determines which underlying architectures are available through a wrapper driver lib, that we developed as run-time support (see later in Section 4.2.4). After that, the *Analyzer* disassembles all given assemblies looking for *Kernel*-annotated methods. This is performed by exploiting reflection capabilities of the CLR (recall Section 2.2.4) that implements all those functions defined in Section 2.5.2. At this point there are two possible scenarios: either no special-purpose architectures are available or there is at least one GPU. In the former case, *Kernel* will be forced to run on a CPU, whereas in the latter case the *Analyzer* performs the first step in compilation toolchain that ends with the execution of a *Kernel* on a GPU.

Kernel execution on a CPU

If no special-purpose architectures are available, the *Analyzer* performs the same steps already listed for the *Map* method in listing 2.9, but on data streams defined in Section 4.1. In listing 4.4 we provide an example of *Map* on a generic stream.

```
/* Example of delegate definition */
public delegate void StreamDelegate<T,K>(InputStream<T>[] outputs,
    OutputStream<K>[] inputs);

/*
  Map spawns a thread for each processor core. Each thread executes the
  same method, called jobMethod, taking a different part of the input
  data, called inputTh, and returning results in a different part of
  the output data, called outputTh.
*/
public virtual void Map( string method_name,
                         OutputStream<T>[] outputs,
                         InputStream<K>[] inputs ) {

      1.  StreamDelegate<T,K> jobMethod = (StreamDelegate<T,K>)
          Delegate.CreateDelegate(typeof(StreamDelegate<T,K>),
          _currInstance, method_name);
      2.  Scatter input and output parameters between _cores threads,
          inputTh and outputTh
      3.  For each thread do
      3.1      jobMethod(outputTh, inputTh);
      4.  Gather output results
}
```

Listing 4.4: C# pseudo-code executed by the Analyzer to perform a computation on a CPU.

Specifically, the *Analyzer* creates a thread pool (TP) of size equal to the number of CPU cores. It scatters data to, and waits to gather the results from the TP.

MSIL *Kernel* code analysis

Having at least one GPU, for each *Kernel*-annotated method found in each class in a given assembly, the *Analyzer* checks whether it is already compiled or not, i.e. whether an executable file with that *Kernel* exists or not. In the former case, the *Analyzer* executes the same steps of the *Map* method in listing 4.2. In the latter case, the *Analyzer* performs the following steps:

1. for each class c

   (a) for each method $m \in met_{CLR}(c)$, if $isKernelAttr_{CLR}(m)$ then

      i. for each stream $\in$ params$_{CLR}$(m), registers in the tuple *regIdentifier* are defined and initialized

      ii. call *Parser* to build an abstract syntax tree (AST).

iii. call *Code Generator* with AST as input.

In order to scan and reconstruct the stack behavior of a method the *Analyzer* leverages the *CLIFileRW* library [108] capabilities.

CLIFileRW library. It is a library specifically designed to read and rewrite .NET binaries. It interacts with .NET reflection only under explicit request so large code bases can be analyzed using it. It is built using memory mapping to avoid unnecessary memory allocation, data is accessed directly on the disk and CLR meta-data tables are exposed as a set of tables using indexers. *CLIFileRW* provides the *ILCursor* class, that is a linear cursor into a stream of CIL instructions. The cursor provides a number of facilities that are needed to generate optimized streaming code, one of which is the abstract interpretation of the operands stack that is particularly important since we need to reconstruct the stack behavior to translate a *stack-based* VM (e.g. CLR) code to a *register-based* VM (CAL IL or PTX) one. Indeed, knowing the evaluation stack height it is possible to know the maximum number of registers required for each statement.

4.2.3 Parser

The main purpose of the *Parser* is to build a AST having a MSIL method m as input. It computes the following steps:

1. for each instruction $instr \in \eta_{CLR}(m)$

   (a) creates an abstract interpretation of *instr* with all information required to the *Code Generator*, and pushes it onto the abstract stack, updating the AST.

```
/* sum - MSIL code (source) */
.method public hidebysig instance void Add() cil managed {
  .custom instance void [CompileLib]StreamDefs.KernelsAttribute::.ctor
     ()
  .maxstack 2
  .locals init (
    [0] int32 a,
    [1] int32 b,
    [2] int32 c)
//...

L_0005: ldloc.0
L_0006: ldloc.1
L_0007: add

//...
}
```

```
/* Parse tree generation */
private void ILCodeScan(...) {
  switch (op) {
    case OpCode.Ldloc: {
        NodeConst node = new NodeConst( _ptxCode, labe, new OperandInfo
            (cil.op, TypesMapping));
        NodesStack.Push(node);}
      break;
    case OpCode.Add:
    case OpCode.Sub:
    case OpCode.Div: {
        Node operand1 = NodesStack.Pop();
        Node operand2 = NodesStack.Pop();
        NodeCALC calcNode = new NodeCALC ( _ptxCode, labe, (int)_cursor
            .Position, new OperandInfo(cil.op, optype, cil.op.Value));

        calcNode.AddNode(operand1);
        calcNode.AddNode(operand2);
        NodesStack.Push(calcNode); }
      break;
      //...
  }
}
//...

/* MSIL code evaluation */
public string EvalBody(Stream[] outArg, params Stream[] inputArgs) {
  Stack<Node> NodesStack = new Stack<Node>();
  //...
  //...
  int stackHeight = 0;

  for each instruction in the current MSIL code
  {
    opcode = OpCode(instruction);
    int numberOfPush = GetPushOnStack(op);
    int numberOfPop = GetPopOnStack(op);

    ILCodeScan(NodesStack);

    stackHeight += numberOfPush - numberOfPop;

    /* At the end of a statement, the operand stack height is equal to
        zero */
    if ( stackHeight == 0 && NodesStack.Count > 0 )
      GenerateIL(NodesStack);
  }
  //...
}
```

In order to reduce the amount of memory required by the Code Generator, the *GeneratorIL* is called at the end of a statement, that is a sequence of CIL instructions such that the height of the operand stack is zero before the first instruction and it is zero after the last instruction, which should not be a conditional branch.

Having the MSIL code for the sum of two integers as input, the *Analyzer* scans it and, depending on the instruction *OpCode*, it determines which type of Node is pushed onto the stack *NodesStack*, i.e. to the AST. These steps are shown in Figure 4.4.

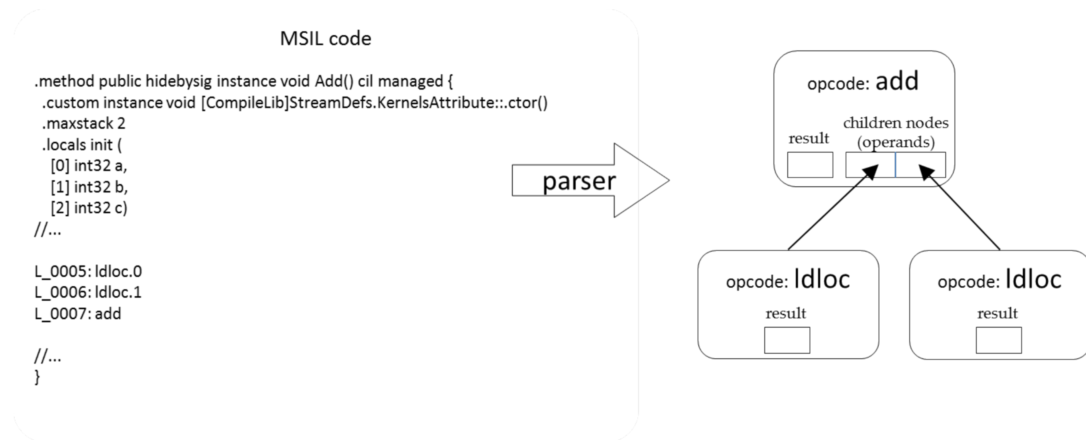

Figure 4.4: **4-Centauri** parser component. Given a MSIL code as input it builds parse tree using Node data structure.

4.2.4 Code Generator

Adopting a postfix traversal, the Code Generator translates each AST node into a PTX instruction, producing an executable program in the end. This code can be stored in an object file or written into memory ready to be executed.

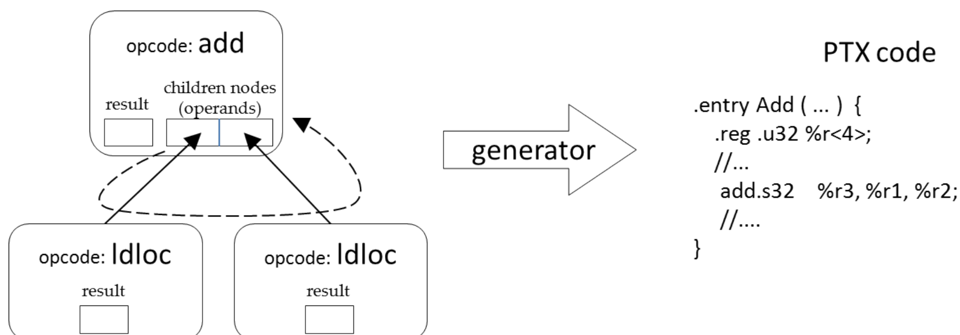

Figure 4.5: **4-Centauri** generator component. Given a parse tree as input generates a PTX code. The visit is postfix.

The translation from CIL op-codes into PTX VM ones is not straightforward because the PTX VM is register based whereas CLR is a stack based VM.

Some CIL (recall Section 2.2.5) instructions are translated into corresponding *register-based* VM instructions, with implicit operands translated to explicit operand registers. The new *register-based* instructions use one byte for the opcode and one byte for each operand register (e.g. *add*).

There are a few exceptions to perform the one-to-one translation:

- Evaluation stack *pop* instructions are translated into `nop` because they are not needed in *register-based* code.
- Instructions related to loading of a local variable onto evaluation stack and storing data from evaluation stack into a local variable are translated into *move* instructions.
- Stack manipulation instructions (e.g. *dup*) are translated into appropriate sequences of *mov* instructions by tracking the state of the evaluation stack.

In the CLI, operands are pushed from local variables onto the evaluation stack before they can be used, and results must be stored from the stack to local variables. Most of these stack `push` and `pop` operations are redundant in our *register-based* run-time as instructions can directly use local variables (registers) without going through the stack. In the generation step, all loads and stores from/to local variables are translated into register *mov* instructions.

A special consideration is required about branch. PTX instructions without a *guard predicate* are executed unconditionally. Predicates are most commonly set as the result of a comparison performed by the *setp* instruction. As an example, consider the high-level code

```
if (i < n)
    j = j + 1;
```

This can be written in PTX as

```
    setp.lt.s32 p, i, n; // p = (i < n)
@p  add.s32 j, j, 1;     // if i < n, add 1 to j
```

To get a conditional branch or conditional function call, use a predicate to control the execution of the branch or call instructions. To implement the above example as a true conditional branch, the following PTX instruction sequence might be used:

```
      setp.lt.s32 p, i, n;    // compare i to n
@!p   bra L1;                 // if false, branch over
      add.s32 j, j, 1;
L1:
```

Driver Wrapper. In order to leverage Nvidia CUDA driver functionalities we have developed a dedicated .NET library, called *NVIDIALib*, based on the .NET platform invocation services. *NVIDIALib* provides a layer of code which translates the CUDA driver *un-managed* C library's interface into a compatible .NET *managed* one in C#.
NVIDIALib implements a set of wrapper types, one for each of CUDA driver fundamental handle type as shown in table 4.2.

CUDA Object	Handle	Wrapper type	Description
Device	CUdevice	Int	CUDA-enabled device
Context	CUcontext	IntPtr	Roughly equivalent to a CPU process
Module	CUmodule	IntPtr	Roughly equivalent to a dynamic library
Function	CUfunction	IntPtr	Kernel
Heap memory	CUdeviceptr	Int	Pointer to device memory

Table 4.2: *NVIDIALib* wrapper types.

Moreover, *NVIDIALib* implements a set of static methods, one for each function exposed by CUDA driver. For instance, the method

```
IntPtr CUDA.InitEnv(int numberOfStreams, int numberOfKernels)
```

wraps the routing that initializes the CUDA environment. Another example is the following routine

```
void cuMemAlloc(CUdeviceptr* dptr, unsigned int bytesize)
```

that allocates *bytesize* bytes of linear memory on the device and returns *dptr* pointer to the allocated memory. This is wrapped by a static method as follows

```
[DllImport(‘‘NVIDIALib.dll’’, ExactSpelling = true, CharSet = CharSet.Auto)]
public static extern int MAllocOnGPU([In, Out] IntPtr dptr, int bytesize);
```

When the CLR's P/Invoke mechanism calls *MAllocOnGPU*, the CLR pins the arguments automatically and unpins them when the unmanaged method returns. However, the GCHandle type must be used explicitly when it is required to pass the address of a managed object to unmanaged code and then, the unmanaged function returns, but unmanaged code might still need to use the object later. For this reason, the following steps are required for each GPU memory allocation

```
GCHandle gInp = GCHandle.Alloc(inputStream.ToArray(), GCHandleType.Pinned);
int ptrInput = CUDA.MAllocOnGPU(job, input_bytesize);
```

All invocation results analysis are performed in the *NVIDIALib.dll* library. Actually, we have not implemented yet an exception management system. We are still working on that.

4.2.5 Runtime support

GPU IL Loader loads an object file to be executed. *Thread scheduler*. It provides run-time support on CPU-side for thread scheduling, scattering and gathering of results, synchronization, exception handling, interfacing to external code, etc. *GPU scheduler*. It provides run-time support on GPU-side for *Kernels* scheduling, scattering and gathering of results, synchronization, exception handling, etc. on GPU-side.

Chapter 5

Evaluation

While the previous chapters presented detailed conceptional and technical descriptions of **4-Centauri**, this chapter evaluates its efficiency using some examples. These are evaluated with regards to the reduction in Completion Time (CT). The major goal of every approach that uses GPUs is a significant performance gain compared to the CPU-based implementation. This is evaluated by processing datasets of various sizes and measuring the processing run-time. Please note that these timings exclude any data transfer between the main and the graphics processors. Moreover, it is important to note that we execute tests with and without compilation steps performed by **4-Centauri** to evaluate how much **4-Centauri** weighs on the CT of an application. This work should be considered preliminary, since most of the efforts in developing **4-Centauri** were spent in making the meta-program infrastructure. We expect to provide more examples in different classes of parallel applications also using different languages targeting CLR.

The following test bed has been used for benchmarks: a notebook equipped with Intel Core 2 Duo, 64-Bit processor running at 2.6 GHz, provided with 4 GBytes of memory and running the Microsoft Windows 7 x64 edition Operating System. The graphics hardware was always Nvidia's GeForce G210M: see Table 5.1 for its main features.

Feature	**Value**
CUDA Cores	16
Processor clock	1.5GHz
Memory clock	800 MGHz
Memory Config	1GB
Memory interface	64 bit
Total amount of Shared memory	16KB
Warp size	32
Max # threads per block	512
CUDA Compatibility	1.2

Table 5.1: Nvidia GeForce G210M specification: main features.

5.1 Mandelbrot algorithm

Mandelbrot set is used to refer both to a general class of fractal sets and to a particular instance of such a set. In general, a Mandelbrot set marks the set of points in the complex plane such that the corresponding Julia set is connected and not computable. "The" Mandelbrot set is the set obtained from the quadratic recurrence equation:

$$z_{n+1} = z_n^2 + C$$

with $z_0 = C$, where points C in the complex plane for which the orbit of z_n does not tend to infinity are in the set. Setting z_0 equal to any point in the set that is not a periodic point gives the same result.

5.1.1 Implementation details and results

We have implemented a parallel version of the Mandelbrot algorithm adopting the data-parallel pattern *Map*. Let *MandelbrotClass* be the class that implements our version of Mandelbrot algorithm. This class has a set of instance fields used to obtain a particular Mandelbrot set; these fields will be mapped on the GPU shared memory. Let *Mandel* be the *Kernel* annotated method that computes a new Mandelbrot set. It refers a set of memory resources. Using notation in Section 2.5.2, *Mandel* $\in \mathbb{MET}_{CLR}$. *Mandel* accesses to $mlocals_{CLR}(Mandel) = \{row, col, x, y, x0, y0, i, xtemp\}$ local variables, $ifields_{CLR}(Mandel) = \{samplewidth, sampleheight, imagewidth, imageheight, offsetX, offsetY, maxiteration\}$ instance fields. It has three parameters, $params_{CLR}(Mandel) = \{rows, cols, results\}$. The first two arguments are *InputStream* typed, whereas the last is *OutputStream* typed. A C# implementation of *MandelbrotClass* is provided in listing 5.1.

```
public class MandelbrotClass
{
  int samplewidth;
  int sampleheight;

  int imagewidth;
  int imageheight;

  int offset_X;
  int offset_Y;

  int maxiteration;

  public MandelbrotClass(int sampleWidth, int sampleHeight, int
     imageWidth, int imageHeight, int offsetX, int offsetY) { ... }

  [Kernel]
  public void Mandel(InputStream<float> rows, InputStream<float> cols,
     OutputStream<float> results)
  {
```

```
    int row = rows.Current;
    int col = cols.Current
    float x = (col * samplewidth) / imagewidth + offset_X;
    float y = (row * sampleheight) / imageheight + offset_Y;
    float x0 = x
    float y0 = y
    for ( int i = 0; i < maxiteration; i++)
    {
      if ( x * x + y * y >= 4) {
        results.Current = (((i % 255f) + 1);
        return;
      }
      float xtemp = x * x - y * y + x0;
      y = 2 * x * y + y0;
      x = xtemp;
    }
  }
}
```

Listing 5.1: C# implementation of Mandelbrot algorithm adopting the stream programming model.

In the host application, we instantiated the *GPUDataParallel* type (see Section 4.1.1, listing 4.2) and invoked its *Map* method passing *MandelKernel* method name as argument. For our experiments we have considered images of different sizes. Figure 5.1 shows Mandel *Kernel* execution CT on different processors, CPU and GPU, using both **4-Centauri** and a Nvidia CUDA implementation of Mandel, and table 5.2 report all timings values obtained.

Image size (n)	CT CPU (s)	CT GPU - 1st run (s)	GPU - 2nd run (s)	CUDA 3.2 (CC 1.2)
64	0.054	0.51	0.231	0.27
128	0.121	0.803	0.477	0.39
256	1.283	1.329	1.064	0.94
512	5.23	2.236	1.926	2.01
1024	15.32	5.16	4.852	4.75
2048	50.822	8.317	8.015	7.793

Table 5.2: Comparison of completion times executing the Mandelbrot *Kernel* on a CPU dual-core and on a GPU. In order to evaluate the overhead introduced by **4-Centauri**, the CT is computed twice, i.e. in the second run no compilation is performed. Moreover, we compared the **4-Centauri** compiled *Kernel* with a Nvidia CUDA implementation of it.

When compilation is done, the CT augments by an average of 298.3 ms. Since the compilation does not depend on input data size, this overhead is constant over different runs. Comparing the results between CPU and GPU CT, we can see that the CPU is faster while input has a dimension of 32Kb or lower without compilation, and of 128Kb with compilation. This is due to data transfer times between CPU and GPU.

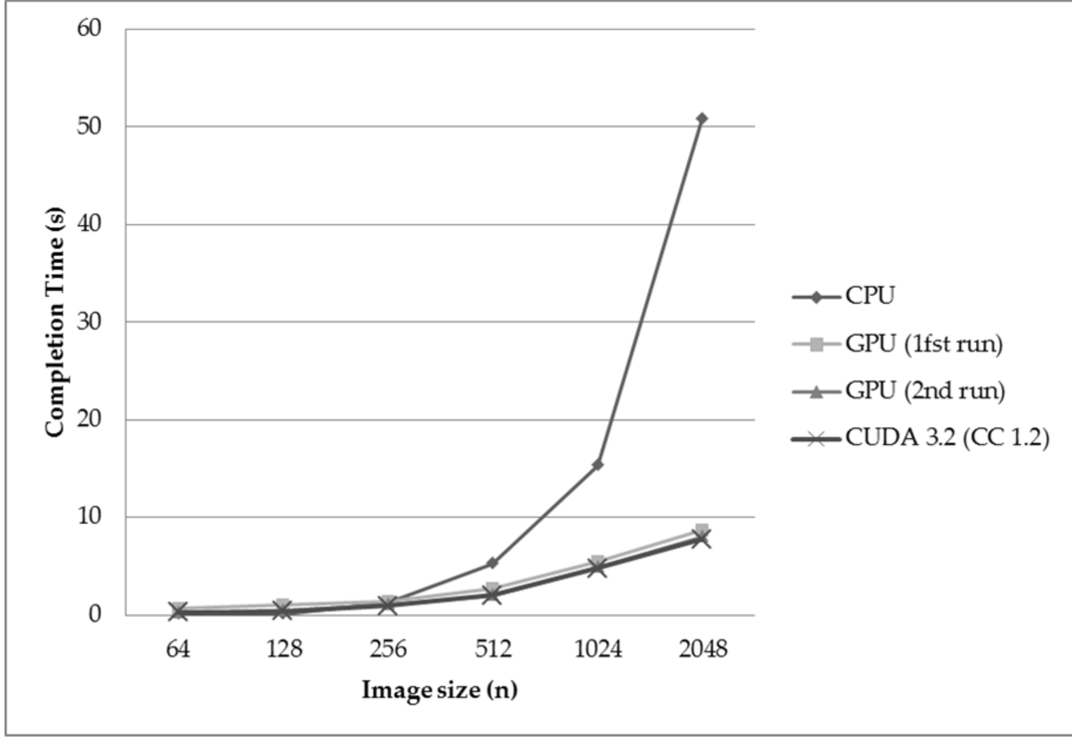

Figure 5.1: Comparison of completion times executing the Mandelbrot *Kernel* on a CPU dual-core and on a GPU. In order to evaluate the overhead introduced by **4-Centauri**, the CT is computed twice, i.e. in the second run no compilation is performed. Moreover, we compared the **4-Centauri** compiled *Kernel* with a Nvidia CUDA implementation of it.k

5.2 Mersenne Twister algorithm

The Mersenne Twister (MT) is a pseudorandom number generator algorithm developed by Matsumoto and Nishimura [97, 98, 109]. Let x and a be word vectors which are w-dimensional row vectors over the two-element field $\mathbb{F}_2 = 0, 1$, identified with machine words of size w (32 bit in our implementation). The MT generates a sequence of word vectors, which are considered to be uniform pseudorandom integers between 0 and $2^w 1$. Dividing by $2^w 1$, each word vector can be a real number in [0, 1]. For a word x with w bit width, it is expressed as the linear recurrence relation:

$$x_{k+n} = x_{k+m} \oplus (x_k^u \mid x_{k+1}^l) \bullet A$$

where n is the degree of the recurrence, m a fixed positive integer, with $1 \leq m \leq n$, $\mid$ as the bitwise or and $\oplus$ as the bitwise exclusive or (XOR), $(x_k^u \mid x_{k+1}^l)$ is a concatenation of r most significant bits of x_k and $w - r$ least significant bits of x_{k-1}, A is a constant $w \times w$ matrix with entries in $\mathbb{F}_2$. Moreover, $(x_k^u \mid x_{k+1}^l) \bullet A$ is the multiplication of the concatenated bit vector by bit matrix A.

Matrix A is chosen for simplicity of computations in the form of

$$x \bullet A = x_{w-1}, x_{w-2}, \cdots, x_0 \bullet \begin{vmatrix} 0 & I_{w-1} \\ a_{w-1} & (a_{w-2}, \cdots, a_0) \end{vmatrix}$$

with I_{n-1} as the $(n-1)(n-1)$ identity matrix (and in contrast to normal matrix multiplication, bitwise XOR replaces addition).

For the special form of matrix A, assuming w is less than or equal to machine word size, $x \bullet A$ multiplication can be efficiently implemented on existing hardware. In order to improve the distribution properties, each generated word is multiplied by a special $w \times w$ invertible transformation matrix from the right: $x \to z : x \bullet T$. A tempering matrix T is chosen so that $x \bullet T$ multiplication, similarly to $x \bullet A$, can be efficiently implemented with bitwise operations:

```
z = x;
z ^= (z >> u);
z ^= (z << s) & b;
z ^= (z << t) & c;
z ^= (z >> l);
```

where u, s, t, l are integer numbers, b and c are suitable bit masks of word size w, and <<,>> as the bitwise left and right shifts, and & as the bitwise *and*. According to the MT2203 [98], values for these coefficients are:

- $(w, n, m, r) = (32, 624, 397, 31)$
- a = 9908B0DF16
- $u = 11$
- (s, b) = (7, 9D2C568016)
- (t, c) = (15, EFC6000016)
- $l = 18$

5.2.1 Implementation details and results

In order to execute MT on a GPU, it is required to have a parallel version of the code listed in 2.8. Considering to execute multiple MT generators in parallel, even "very different" (by any definition) initial state values do not prevent the emission of correlated sequences by each generator sharing identical parameters. To solve this problem and to enable efficient implementation of MT on parallel architectures, a special off-line library, named SIMD-oriented Fast MT (SFMT), for the dynamic creation of MT parameters, was developed by Matsumoto and Nishimura in [98]. In Listing 5.2 we implements our version of the SFMT using C# and **4-Centauri** types and metadata.

```
public class MTClass
{
  /* Period parameters */
  private const int N = 624; // 624 length array to store the state of
      the generator
  private const int M = 397;

  /* the array for the state vector  */
  private readonly int[] _mt = new UInt32[N];
  private int _mti;

  private static readonly int[] _mag01 = { 0x0, 0x9908b0df };

  [Kernel]
  public int MTRandomGenerate(InputStream<int> ds_MT, InputStream<int>
     seed, OutputStream<int> results)
  {
    int iState, iState1, iStateM, iOut;
    unsigned int mti, mti1, mtiM, x;
    unsigned int mt[19], matrix_a;

    //Load bit-vector A
    matrix_a = ds_MT.Current;

    //Initialize current state
    mt[0] = seed.Current;
    for (iState = 1; iState < 19; iState++)
        mt[iState] = (1812433253U * (mt[iState - 1] ^ (mt[iState - 1]
           >> 30)) + iState) & 0xFFFFFFFFU;

    iState = 0;
    mti1 = mt[0];
    for (iOut = 0; iOut < nPerRng; iOut++) {
        iState1 = iState + 1;
        iStateM = iState + 9;
        if(iState1 >= 19) iState1 -= 19;
        if(iStateM >= 19) iStateM -= 19;
        mti  = mti1;
        mti1 = mt[iState1];
        mtiM = mt[iStateM];

        // MT recurrence
        x = (mti & 0xFFFFFFFEU) | (mti1 & 0x1U);
        x =  mtiM ^ (x >> 1) ^ ((x & 1) ? matrix_a : 0);

        mt[iState] = x;
        iState = iState1;

        //Tempering transformation
        x ^= (x >> 11);
        x ^= (x << 7) & 0x9d2c5680;
```

```
            x ^= (x << 15) & 0xefc60000;
            x ^= (x >> 18);

            //Convert to (0, 1] float and write to global memory
            results.Current = ((float)x + 1.0f) / 4294967296.0f;
        }
    }
}
```

Listing 5.2: C# implementation of SIMD-oriented Fast Mersenne Twister algorithm adopting the stream programming model.

In the host application, we instantiated the *GPUDataParallel* type (see Section 4.1.1, listing 4.2) and invoked its *Map* method passing *MTRandomGenerate* method name as argument. For our experiments we have considered to generate different amount of samples. Figure 5.2 shows MTRandomGenerate *Kernel* execution CT on different processors, CPU and GPU, using both **4-Centauri** and a Nvidia CUDA implementations of it, and table 5.2 report all timings values obtained.

Figure 5.2 shows MTRandomGenerate *Kernel* execution CT on different processors, CPU and GPU, using both **4-Centauri** and a Nvidia CUDA implementation of MT, and table 5.3 report all timings values obtained.

#Samples (x10^6)	CT CPU (ms)	CT GPU - 1st run (s)	GPU - 2nd run (s)	CUDA 3.2 (CC 1.2)
1	45.18	4.13	1.83	1.53
2	86.82	6.01	3.21	2.97
5	214.1	10.21	7.99	7.29
10	431.4	17.44	15.37	14.48
15	638.9	23.84	22.04	21.67
24	1064.4	37.83	35.12	34.62

Table 5.3: Comparison of completion times executing the Mersenne-Twister *Kernel* on a CPU dual-core and on a GPU. In order to evaluate the overhead introduced by **4-Centauri**, the CT is computed twice, i.e. in the second run no compilation is performed. Moreover, we compared the **4-Centauri** compiled *Kernel* with a Nvidia CUDA implementation of it.

As for the Mandelbrot test, the step of **4-Centauri** compilation increases the CT. In this case the overhead due to the compilation is by an average of 2.25 ms. Since the compilation does not depend on input data size, this overhead is constant over different runs. Comparing the results between CPU and GPU CT, we can see that the CPU is always slower than GPUs because of limited number of data transfers between CPU and GPU.

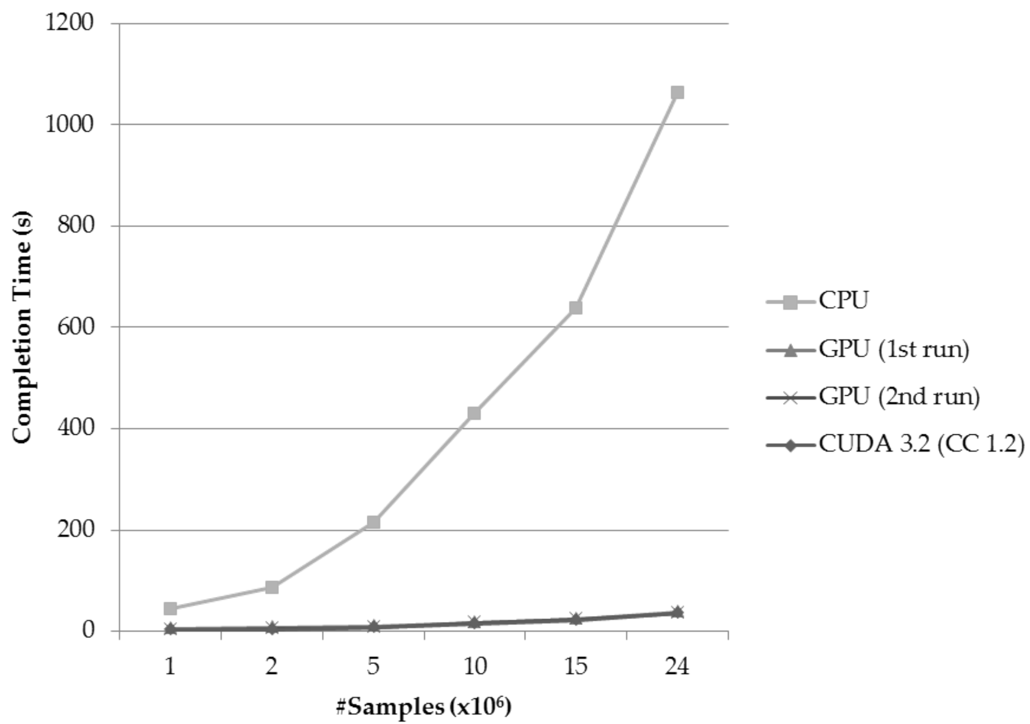

Figure 5.2: Comparison of completion times executing the Mersenne-Twister *Kernel* on a CPU dual-core and on a GPU. In order to evaluate the overhead introduced by **4-Centauri**, the CT is computed twice, i.e. in the second run no compilation is performed. Moreover, we compared the **4-Centauri** compiled *Kernel* with a Nvidia CUDA implementation of it.

Conclusions

C.1 Thesis summary

In this thesis we addressed the problem of exposing non-conventional computing devices to VEE programs without changing the VEE base definition. We first introduced the ASM formalism we used to model parallel programming paradigms, and then the Common Language Infrastructure that we used for our study (due to obvious similarities the JVM could have been used and we consider our results to be naturally extended to this platform). We discussed how the essential traits of these paradigms can be mapped onto CLI semantic entities so that program transformation will benefit from this additional information. We finally presented **4-Centauri**, a system for transforming *Kernels* expressed using CLI intermediate language into GPU IL code (in particular Nvidia PTX).

In our study we followed two research lines: we realized a meta-program capable of compiling VEE intermediate language into GPU code as a proof of the fact that it is actually possible to expose a very peculiar co-processor, such as the GPU as a resource accessible to managed programs. We also discussed a more general approach for the introduction of hints about parallel execution into VEE source programs; in our design the programmer annotates code to provide more information about possible parallel execution that can be exploited by meta-programs or in general by the underlying compilation infrastructure.

A distinctive trait of our approach lies in the hinting nature of annotations, leaving the task of their exploitation to the execution infrastructure. We see two main benefits for this approach: users can rely on more abstract and stable notions instead of having to follow the continuous changing of real hardware; code generators can adapt annotated programs to different architectures and changing configurations with a little overhead of Just in time compilation that has already proven to be viable in this context.

Another important trait is the idea of leveraging on already existing boundaries of semantic elements of a STEE to express significant boundaries of parallel concurrent paradigms. We followed a widely adopted pattern by STEEs programming frameworks in which programmers are asked to develop a module with a specific interface M whose execution is orchestrated by an orchestrator O; the resulting

program is *O(M)* where the user program is invoked from outside as is the case, for instance, with Web applications in Java or .NET. In our case, module M is annotated with parallel-related annotations that may be used by O, which is responsible for its execution. We may even think of different orchestration modules that can be used without any change to the source program (for instance a sequential execution in order to simplify functional debugging).

Our work has shown that meta-data are expressive enough to support program transformation towards processing units supporting explicit parallelism, as GPUs, and that all the models available in literature can be represented. The implementation of a transformation system for the relevant case of GPUs testified the viability of the approach in practice.

The **4-Centauri** system was originally intended as a practical proof of the possibility to expose the full programmability of GPUs to managed programs without needing to apply any changes to the virtual execution environment. In this respect, it may be considered a managed counterpart of CUDA, Brook+, or OpenCL environments which allows to schedule general purpose code to the GPU subsystem. However, we found that the **4-Centauri** scheduler has the unique capability, given a model of costs, of deciding at run-time whether to schedule a program on the CPU or GPU depending on the nature of input data (for instance, whether it is more convenient to pay for data transfer to the GPU memory and perform GPU computations, or the communications overhead is such that traditional execution is to be preferred).

C.2 Suggestions for Future Research

Although **4-Centauri** shows that a mapping is possible between managed code and GPU code, the compiler is still in a prototype state and many research and development directions can be taken. First of all, the back-end of the compiler should be extended to target the AMD IL used in the AMD GPUs, which is analogous to Nvidia PTX. Moreover, it is possible to map more abstract concepts, such as structured data types, onto the GPU code. Our work contributes to answer the question of whether a redesign of VEEs is required or not, consequently further work in the performance direction is now needed. **4-Centauri** can be greatly improved in performance, and there is room to develop new optimization techniques for this special-purpose hardware that may eventually join the CPU in a single unit as proposed by AMD. We are also interested in studying the set of algorithms that can be expressed efficiently through our abstractions and possibly extend the set of annotations. A very important research direction is the extension of the approach to more non-conventional processing units, such as Cell BE (or in future Larrabee), in order to understand if the approach can be generalized in practice, and in particular if the mapping between programming abstractions of CLI and the particular processing units is really helpful for the programmer.

Bibliography

[1] E. International, *Standard ECMA-335 - Common Language Infrastructure CLI*, 4th ed., June 2006. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-335.htm

[2] J. Manferdelli, N. Govindaraju, and C. Crall, "Challenges and Opportunities in Many-Core Computing," in *IEEE*, vol. 96, may 2008, pp. 808–815.

[3] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech. Rep. UCB/EECS-2006-183, december 2006.

[4] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 2007.

[5] M. Horowitz and W. Dally, "How scaling will change processor architecture," in *Proc. IEEE International Conference on Solid-State Circuits*, 2004, pp. 132–133.

[6] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, March 2007.

[7] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K.Yazawa, "The design and implementation of a first-generation Cell processor," in *Proceedings of the Custom Integrated Circuits Conference*, 2005.

[8] L. Seiler, D. Carmean1, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. S. R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A Many-Core x86 Architecture for Visual Computing," in *ACM Trans. Graph. 27, 3, Article 18*, august 2007.

[9] J. Henning, "SPEC CPU2000: measuring CPU performance in the new millennium," *IEEE Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[10] F. Pellacini, K. Vidimče, A. Lefohn, A. Mohr, M. Leone, and J. Warren, "Lpics: a hybrid hardware-accelerated relighting engine for computer cinematography," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 464–470, 2005.

[11] H. Nguyen, *GPU Gems 3*. Addison-Wesley Professional, 2007.

[12] R. Dumont, F. Pellacini, and J. Ferwerda, "A perceptually-based texture caching algorithm for hardware-based rendering," in *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*. London, UK: Springer-Verlag, 2001, pp. 249–256.

[13] J. Lengyel, M. Reichert, B. Donald, and D. Greenberg, "Real-time robot motion planning using rasterizing computer graphics hardware," *Computer Graphics (SIGGRAPH'90 Proceedings)*, vol. 24, pp. 327–335, 1990.

[14] G. Kedemand and Y. Ishihara, "Brute force attack on UNIX passwords with SIMD computer," *USENIX Security Symposium (SECURITY'99 Proceedings)*, pp. 93–98, 1999.

[15] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha, "Fast computation of generalized Voronoi diagrams using graphics hardware," *Computer Graphics (SIGGRAPH'99 Proceedings)*, pp. 277–286, 1999.

[16] S. Upstill, "The RenderMan companion," 1990.

[17] M.S.Peercy, M.Olano, J.Airey, and P.J.Ungar, "Interactive multi-pass programmable shading," *Computer Graphics (SIGGRAPH'00 Proceedings)*, vol. 34, pp. 425–432, 2000.

[18] W. Mark, R. Glanville, K.Akeley, and M.J.Kilgard, "Cg: A system for programming graphics hardware in a C-like language," *Computer Graphics (SIGGRAPH'03)*, vol. 37, pp. 896–907, 2003.

[19] NVIDIA Corp., "NVIDIA CUDA Programming Guide," NVIDIA Corporation, Tech. Rep. 3.2, September 2010.

[20] AMD Corp., "AMD Stream Computing. Compute Abstraction Layer (CAL) Technology. Programming Guide," AMD Corporation, Tech. Rep. 2.2, march 2010.

[21] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," in *Proc. ACM SIGGRAPH 2004*, 2004, p. 777.

[22] C. Kessler, *Automatic parallelization, New approaches to Code Generation, data distribution, and performance prediction*, ser. Vieweg Advances Studies in Computer Science. Verlag Vieweg, 1994.

[23] T. Brandes, “Adaptor. Users guide,” Institute for Algorithms and Scientific Computing. FhG, Tech. Rep., 2004.

[24] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel, *The High Perfomance Fortran handbook.* MIT Press, 1997.

[25] “The SUIF 2 compiler system,” 1999. [Online]. Available: http://suif.stanford.edu/suif/suif2/doc-2.2.0-4/

[26] N. Travinin, H. Hoffmann, R. Bond, H. Chan, J. Kepner, and E. Wong, “pMapper: Automatic Mapping of Parallel Matlab Programs,” in *Users Group Conference*, 2005, pp. 254– 261.

[27] L. Snyder, *A Programmers Guide to ZPL.* Department of Computer Science and Engineering, University of Washington, 1999.

[28] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.

[29] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar, “X10: An object-oriented approach to non-uniform clustered computing,” in *OOPSLA*, 2005.

[30] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. Maessen, S. Ryu, G. S. Jr., and S. Tobin-Hochstadt, “The Fortress Language Specification,” Sun Microsystems, Inc., Tech. Rep., 2007. [Online]. Available: http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf

[31] P. Pacheco, *Parallel programming with MPI.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[32] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification 2.0*, 1999. [Online]. Available: http://java.sun.com/docs/books/jvms

[33] *.NET Framework 4.0*, 2010. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms229335.aspx

[34] R. Whalley, A. Petitet, , and J. Dongarra, “Automated empirical optimization of software and the ATLAS project,” *Parallel Comput.*, vol. 27, pp. 3–35, 2001.

[35] Z. Chen, J. Dongarra, P. Luszczek, and K.Roche, “Self adapting software for numerical linear algebra and LAPACK for clusters,” *Parallel Comput.*, p. 17231743, 2003.

[36] M. Frigo, “A fast Fourier transform compiler,” in *Proc. PLDI*, 1999.

[37] D. Tarditi, S. Puri, and J. Oglesby, “Accelerator: Using data-parallelism to program GPUs for general purpose uses,” in *12th Int. Conf. Architect. Support Program. Lang. Oper. Syst.*, 2006, p. 325.

[38] J. Larusand and R. Rajwar, “Transactional Memory,” 2006.

[39] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, *OpenMP Application Program Interface.* OpenMP Architecture Review Board, 2008.

[40] L. R. Group, *Mentat Programming Language.* Department of Computer Science. University of Virginia, 1998.

[41] M. Aldinucci, M. Danelutto, and P. Dazzi, “MUSKEL: an expandable skeleton environment,” in *Scientific International Journal for Parallel and Distributed Computing*, vol. 8, 2007, pp. 325–341.

[42] M. Monteyne, “Rapidmind multi-core develpment platform,” RapidMind, Tech. Rep., 2007.

[43] M. Segal and K. Akeley, “The OpenGL Graphics System: A specification,” 2010.

[44] Microsoft, “DirectX Developer Center,” 2010, http://msdn.microsoft.com/en-us/directx/default.aspx.

[45] G. Kiczales, “Aspect-oriented programming,” *ACM Comput. Surv.*, p. 154.

[46] R. Goldberg, “Survey of Virtual Machine Research,” *Computer*, pp. 34–35, June 1974.

[47] J. Smith and R. Nair, “The architecture of Virtual Machines,” *IEEE Computer*, vol. 38, no. 5, pp. 32–38, May 2005.

[48] K. Gabor and S. Janos, “A Model-Based Approach to Self-Adaptive Software,” *IEEE Intelligent Systems*, vol. 14, pp. 46–53, 1999.

[49] P. Sestoft, “Runtime Code Generation with JVM and CLR,” Department of Mathematics and Physics, IT University of Copenhagen, Denmark, Tech. Rep., 2002.

[50] L. Valiant, “A bridging model for parallel computation,” *Communication of ACM*, vol. 33, pp. 103–111, 1990.

[51] A. Burks, H. Goldstine, and J. von Neumann, “Preliminary discussion of the logical design of an electronic computing instrument,” The Institute of Advance Study, Princeton, Report to the U.S. Army Ordnance Department, Tech. Rep.

[52] J. Aho, J. Hopcroft, and J. Ullman, *The design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[53] D. Skillicorn, "Practical parallel computation," Department of Computing and Information Science, Queen's University, Kingston, Ontario, Tech. Rep. ISSN-0836-0227-91-312, 2001.

[54] W. McColl, "An architecture independent programming model for scalable parallel computing," in *General Purpose Parallel Computing.* British Computer Society Parallel Processing Specialist Group, december 1993, pp. 1–17.

[55] E. Börger, "The ASM ground model method as a foundation for requirements engineering," *Verification: Theory and Practice*, vol. 2772, pp. 145–160, 2003.

[56] E. Börger and R. Stärk, "Abstract State Machines: A Method for High-Level System Design and Analysis," in *Springer-Verlag*, 2003.

[57] E. Börger, "The origins and the development of the ASM method for high level system design and analysis," *Journal of Universal Computer Science*, vol. 8, pp. 2–74, 2002.

[58] Y. Gurevich, "Evolving Algebras: An attempt to discover semantics," in *G. Rozenberg, A. Salomaa eds, Current Trends in Theoretical Computer Science*, 1993, pp. 266–292.

[59] D. K. G. Campbell, "A survey of models of parallel computation," University of York, Dept. of Computer Science, Tech. Rep., 1997.

[60] Z. Yunquan, C. Guoliang, S. Guangzhong, and M. Qiankun, "Models of parallel computation: a survey and classification," in *Front. Comput. Sci. China*, 2006, pp. 156–165.

[61] S. Fortune and J. Wyllie, "Parallelism in random access computers," in *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, 1978, pp. 114–118.

[62] R. Cole and O. Zajicek, "The APRAM: incorporating asynchrony into the PRAM model," in *Proceedings of the 1st Annual ACM SPAA'89*, 1989, pp. 169–178.

[63] P. Gibbons, "A more practical PRAM model," in *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, 1989, pp. 158–168.

[64] K. Mehlhorn and U. Vishkin, "Randomized and deterministic simulations of PRAMs by parallel computers with restricted granularity of parallel memories," in *Acta Informatica*, 1984, pp. 339–274.

[65] A. Aggarwal and A. Chandra, "Communication complexity of PRAMs," in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, vol. 317. Springer Berlin / Heidelberg, 1988, pp. 1–17.

[66] A. Aggarwal, A. Chandra, and M. Snir, "On communication latency in PRAM computations," in *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '89. ACM, 1989, pp. 11–21.

[67] P. de la Torre and C. Kruskal, "Towards a single model of efficient computation in real parallel computers." in *Elsevier Science Publishers*, 1992, pp. 395–408.

[68] T. Heywood and S. Ranka, "A practical hierarchical model of parallel computation I. The model." *Journal of Parallel and Distributed Computing*, vol. 3, no. 16, pp. 212–232, 1992.

[69] P. Hatcher, A. Lapadula, M. Quinn, and R. Anderson, "Compiling Data-Parallel Programs for MIMD Architecture," *Parallel Computing: from theory to sound practice, in W. Joosen and E. Milgrom editors*, 1992.

[70] L. Valiant, "Bulk-Synchronous Parallel computers," Aiken Computation Laboratory, Harvard University, Tech. Rep. TR-08-89, 1989.

[71] J. Hill, B. McColl, and D. Stefanescu, "BSPlib: the BSP Programming library," *Parallel Computing*, pp. 1947–1980, 1998.

[72] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," in *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993, pp. 1–12.

[73] W. Peterson and D. Brown, "Cyclic Codes for Error Detection," *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–235, jan. 1961.

[74] NVIDIA Corp., "NVIDIA's Next Generation CUDA Compute Architecture," NVIDIA Corporation, Tech. Rep., 2010.

[75] R. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal*, vol. 26, no. 2, pp. 147–160, 1950.

[76] P. Nelson, "A comparison of PASCAL intermediate languages," in *Proceedings of the 1979 SIGPLAN symposium on Compiler construction*, ser. SIGPLAN '79. ACM, 1979, pp. 208–213.

[77] D. Schleicher and R. Taylor, "System overview of the application system/400," vol. 38. IBM Corp., June 1999.

[78] J. Sugerman, G. Venkitachalam, and B. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. USENIX Association, 2001, pp. 1–14.

[79] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfiled., "Xen and the art of virtualization," in *Proceedings of the SOSP*, 2003.

[80] R. Pickering, *Foundations of F#*. Apress, 2007.

[81] *Objective Caml*, INRIA, 2009. [Online]. Available: http://caml.inria.fr/ocaml/

[82] *Python Programming Language*, Python Software Foundation, 2009. [Online]. Available: http://www.python.org

[83] *The Parrot VM Design Document*, 2009. [Online]. Available: http://docs.parrot.org/parrot/latest/html/pdds.html

[84] NVIDIA Corp., "NVIDIA Compute. PTX: Parallel Thread Execution," NVIDIA Corporation, Tech. Rep. 2.0, january 2010.

[85] AMD Corp., "AMD Stream Computing. Compute Abstraction Layer (CAL) Technology. Intermediate Language (IL)," AMD Corporation, Tech. Rep. 2.0b, march 2010.

[86] M. de Icaza, *Mono Project*, 2009. [Online]. Available: http://mono-project.com

[87] A. Cisternino, *Multi-Stage and Meta-Programming Support in Strongly Typed Execution Engines*, ser. PhD Thesis. University of Pisa, 2003.

[88] C. Dittamo, A. Cisternino, and M. Danelutto, "Parallelization of *C#* programs through annotations," in *International Conference on Computational Scienc, ICCS 2007*, 2007, pp. 585–592.

[89] S. Lidin, *Inside Microsoft .NET IL assembler*. Microsoft Press, 2002.

[90] W. Cazzola, A. Cisternino, and D. Colombo, "Freely annotating *C#*," *Journal of Object Technology*, vol. 4, no. 10, pp. 31–48, 2005.

[91] *.NET Framework Class Library, COM interop and platform invoke services*, 2010. [Online]. Available: http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.aspx

[92] S. McLean, J. Naftel, and K. Williams, *Microsoft .NET Remoting*. Microsoft Press, 2003.

[93] S. Toub, "Patterns of parallel programming," Microsoft Corporation, Tech. Rep., 2010.

[94] D. Leijen and J. Hall, "Optimize managed code for multi-core machines," Microsoft Corporation, Tech. Rep., 2007.

[95] Microsoft, "Introduction to PLINQ," 2010, `http://msdn.microsoft.com/en-us/library/dd997425.aspx`.

[96] O. Dahl and K. Nygaard, "SIMULA: an ALGOL-based simulation language," *Communication ACM*, vol. 9, no. 9, pp. 671–678, 1966.

[97] M. Makoto and N. Takuji, "Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, 1998.

[98] M. Makoto and N. Takuji, "Dynamic creation of pseudorandom number generators," in *Monte Carlo and Quasi-Monte Carlo Methods*. Springer, 1998, pp. 56–69.

[99] G. Attardi, A. Cisternino, and D. Colombo, "CIL + Metadata > Executable Program," *Journal of Object Technology*, vol. 3, no. 2, pp. 19–26, 2003.

[100] J. Larus and H. Sutter, "Software and the concurrency revolution," *ACM Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[101] M. Wolfe, *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996.

[102] R. Gupta, E. Mehofer, and Y. Zhang, *Profile Guided Compiler Optimization.* chap. 4 in The Compiler Design Handbook Optimization and Machine Code Generation, Y.N. Srikant and P. Shankar, 2002.

[103] S. Akhter and J. Roberts, "Multi-threaded debugging techniques," April 2007, http://www.drdobbs.com/199200938.

[104] G. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *AFIPS Conf. Proc.*, 1967, p. 483485.

[105] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly Media, 1996.

[106] *ATI CTM Guide*, 1st ed., AMD, 2006.

[107] "The OpenCL 1.0 Specification," Khronos OpenCL Working Group, Tech. Rep. 48, 2009.

[108] A. Cisternino, “CLIFileRW,” 2008. [Online]. Available: http://www.codeplex.com/clifilerw

[109] M. Saito, “A variant of Mersenne Twister suitable for graphic processors,” *CoRR*, vol. abs/1005.4973, 2010.