FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Masters' Degree in Informatics Engineering
Dissertation

ÆminiumGPU

A CPU-GPU Hybrid Runtime for the Æminium Language

August 31, 2011

Alcides Fonseca
amaf@student.dei.uc.pt

Advisors at DEI:
Bruno Cabral
Paulo Marques

Abstract

Given that CPU clock speeds are stagnating, programmers are resorting to parallelism to improve the performance of their applications. Although such parallelism has usually been attained using either multicore architectures, multiple CPUs and/or clusters of machines, the GPU has since been used as an alternative. GPUs are an interesting resource because they can provide much more processing power at a fraction of the cost of CPUs.

However, GPU programming is not an easy task. Developers that do not understand the programming model and the hardware architecture of a GPU will not be able to extract all of its processing potential. Furthermore, it is even harder to write code for the GPU that improves the performance compared to an optimized CPU version.

This thesis proposes a high-level programming framework for parallel programs on both CPUs and GPUs. This approach, named ÆminiumGPU, drives inspiration from Functional Programming and currently allows developers to implement programs based on the Map-Reduce pattern. In the future, the framework can be extended with other higher-order functions.

ÆminiumGPU does not force developers to understand the particularities of GPU programming. They write programs in pure Java (and soon Æminium) and specific parts of that code are compiled to OpenCL and executed on the GPU.

In order to generate code with good performance, ÆminiumGPU performs special optimizations for the architecture of GPUs. For instance, it avoids unnecessary compilations and data transfers. Despite these optimizations, programs will not always run faster just by executing them on the GPU. It is possible that CPU code can evidence better performance than GPU versions. To handle such cases and to ensure the fastest version is always executed, ÆminiumGPU automatically decides wether a particular operation should be executed on the GPU or the CPU. These decisions are based on code complexity and input data size, collected at compile-time and run-time.

ÆminiumGPU contributes to reducing the development time and effort required for writing GPU programs. The framework also increases the performance of Java and Æminium code. The contributions of this thesis also include a cost model for reasoning about the fastest architecture for a given program block.

Keywords: GPU, GPGPU, compilers, parallel, multicore, skeleton, OpenCL, parallel

Acknowledgements

The work hereby presented was only possible thanks to both my advisors, Bruno Cabral and Paulo Marques, who welcomed me into the Æminium research group. They have guided me along the course of this thesis, providing valuable feedback and suggestions that have improved the quality of my work and research.

I would also like to thank Jonathan Aldrich, the project lead at Carnegie Mellon University, for welcoming me as a Visiting Researcher during the 3 months I was there, and for discussing this work with me, providing useful feedback.

This work was partially supported by the Portuguese Research Agency FCT, through CISUC (R&D Unit 326/97) and the CMU—Portugal program (R&D Project Aeminium CMU-PT/SE/0038/2008).

Finally, this work would not be possible without the support from my parents, who provided me with the time and means to do it. Additionally, I would also like to thank my colleagues and friends who worked side-by-side with me and helped me by reviewing the document.

Alcides Fonseca

Contents

List of Tables

List of Figures

List of Acronyms

CPU Central Processing Unit

GPU Graphics Processing Unit

GPGPU General Purpose GPU programming

HLSL High Level Shader Language

GLSL OpenGL Shading Language

FP Functional Programming

OO Object-oriented

Listings

1 Introduction

This chapter is organized in six sections. The first section explains the motivation and relevance of the current work. The second section provides background on the challenge of working with the GPU for General Purpose Programming. In the third section, the umbrella project Æminium is introduced to provide the context for this project. The fourth section defines the goals. The fifth section states the contributions of this work. Finally, the last section describes the structure of this document.

1.1 Motivation

CPU speed is no longer increasing according to Moore's Law because of physical factors, such as heat, power consumptions and leakage[32]. The alternative path for manufacturers has been to increase the number of cores inside each processor. This way, CPUs can now process more instructions, but in parallel. On the other hand, parallel programming is not trivial due to concurrency, synchronization and expression of parallelism.

This new architecture is forcing programmers to write concurrent programs in order to take advantage of the full processing power available. This shift is becoming mainstream, and languages such as Æminium, Erlang, Haskell or Fortress are exploring those possibilities.

While the number of cores in a CPU is growing moderately, GPUs already have a much significant number of cores. Commodity graphics cards can have up to 512 cores while common processors are limited to 8 cores. In terms of raw processing power, GPUs can have up to 24 times more throughput. Thus, it comes as no surprise that the GPUs are starting to be used for General Purpose Programming.

A brief comparison of the processing power (in Giga or TeraFLOPS) among

the latest generation Intel Xeon X5560 CPU, NVIDIA Tesla C1060 and ATI Radeon HD 5970 CPUs can be seen in Table 1.1. Notice that the values for the CPU are theoretical and LINPACK benchmarks have only run at a maximum 96% of those values [20]. A more general view of the evolution of processing power of GPU vs CPU can be seen in Figure 1.1, included in NVIDIA's OpenCL Programming Guide[5].

Table 1.1: Processing power across mainstream CPU and GPUs

	Single precision	Double precision
Intel Xeon X5560 3.0GHz [20]	192 GFLOPS	96 GFLOPS
NVIDIA Tesla C1060 [27]	933 GFLOPS	78 GFLOPS
ATI Radeon HD 5970 [7]	4.64 TFLOPS	928 GFLOPS

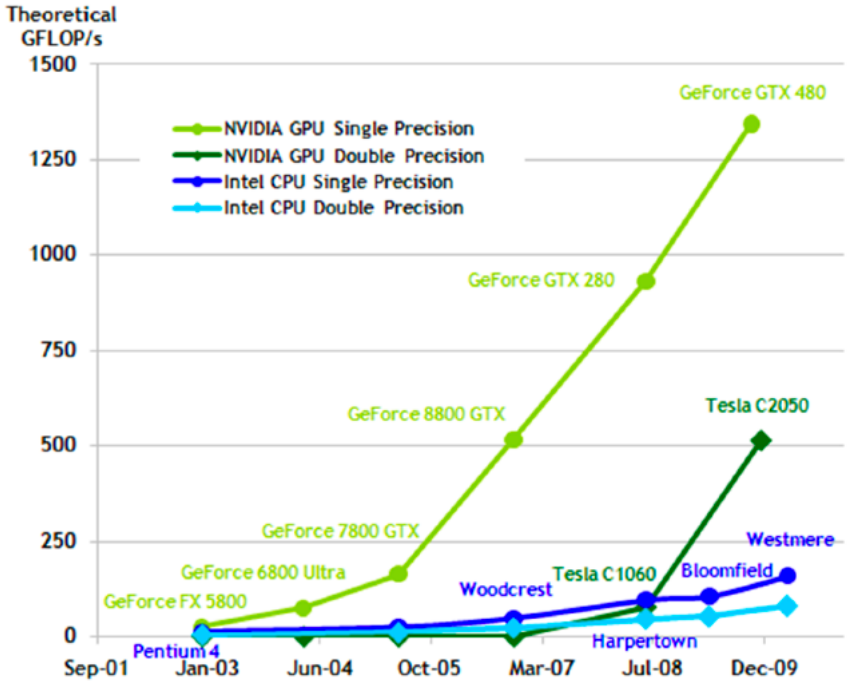

Figure 1.1: Floating-Point operations per second for the CPU and GPU

The architecture of a GPU is inherently different from one of a CPU. GPUs are mainly designed to perform in computations included in the generation of 3D computer graphics, mainly arithmetic operations with vectors and matrixes. As such, GPUs are optimized to process floating-point operations on

large blocks of data at once.

1.2 Challenges of GPGPU programming

Recently there have been efforts to make the GPU processing power available for general-purpose programming[22] (referred to as GPGPU from now on). Low-level APIs such as NVidia's CUDA[3] and OpenCL[5] were developed to provide programmers with a way of running parts of their programs on graphics cards.

These languages have been designed to be as similar as possible to the C programming language in order to reduce the gap between CPU and GPU programming. However, they are not completely similar. There are several differences, many of which inspired by the different nature of the architecture and even by the different models of graphics cards.

A first issue is the selection of which parts of an application should be executed on the GPU. The main requirement for those parts is that they can be executed in parallel.

Secondly, even if the code can be parallelized, there is no guarantee that it will be faster on the GPU. Hence, migrating code to the GPU can be a waste of resources. Factors that influence the performance of a program on the GPU include:

- Memory copy overhead;
- Thread divergence;
- Work unit scheduling overhead;
- Kernel compilation overhead.

Among these, the most important is copying the data that is going to be processed from the main memory to the GPU internal memory. It is essential to decrease the relative importance of that overhead by writing programs that are massively parallel.

GPU cores are organized in groups and, inside each group, they execute the same instructions. If some condition — the `if` instruction for instance — divides the control flow inside the same group, the different flows have to be executed sequentially. This is known as *thread divergence*. Divergence slows down execution on the GPU and must be avoided.

These are only a few examples of how GPU programming is different from traditional CPU programming. But, they provide evidence of how steep the learning curve for GPGPU is.

ÆminiumGPU tackles the challenge of allowing programmers to think about their problems from an higher level. Furthermore, programmers can target GPUs without learning a new programming model or writing any GPU specific code.

Furthermore, even programmers that are already experienced in GPU programming may be more productive by writing their code in a higher-level programming language. Programmers do not have to consider the low-level details of GPU programming, but can still benefit from the performance of GPUs.

1.3 Æminium

This work is part of the Æminium project[33], a collaboration between Carnegie Mellon University, University of Coimbra, University of Madeira and Novabase. The project is entitled *Freeing Programmers from the Shackles of Sequentiality* and focus on providing mainstream programmers with a practical framework for developing massively concurrent applications.

1.3.1 Æminium Programming Language

The main concept of Æminium is that the language is concurrent by default. Programmers are not required to parallelize the code, they simply describe the parts and it is automatically executed in parallel.

This means that the execution of code does not follow the lexicographic order. Instead programmers explicitly annotate methods with *access permissions*. These permissions express wether the method owns or shares the access to a variable.

For example, if two given methods have *full permission* over the variables each one uses, they can be executed in parallel. If two other methods have a *shared permission* over their variables, the execution will have to be sequential. Using these permissions, it is guaranteed that when programs execute concurrently there will be no inconsistencies in shared memory.

Listing 1.1: Example of Æminium parallelization

```
1 Collection students = getStudents();
2 Collection professors = getProfessors();
3 Collection peopleAtSchool = students + professors;
4 printCollection(students);
```

Listing 1.1 represents the type of execution present in Æminium. Since lines 1 and 2 share no variable, they can be executed in parallel and their

order is not guaranteed. The instruction at line 3 will only be executed once the two previous operations are completed because their result is required as input. The forth instruction only depends on the first, and therefore it can be in executed concurrently with the two previous. These dependencies are represented as a graph in Figure 1.2.

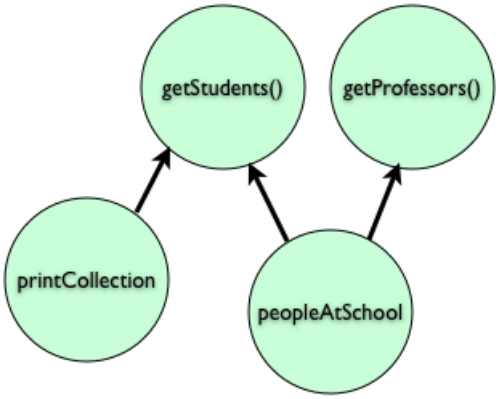

Figure 1.2: Execution dependencies of Æminium parallelization example

1.3.2 Architecture

The language implementation is split in two main components: the Æminium Compiler and the Æminium Runtime. The pipeline that connects them is shown in Figure 1.3.

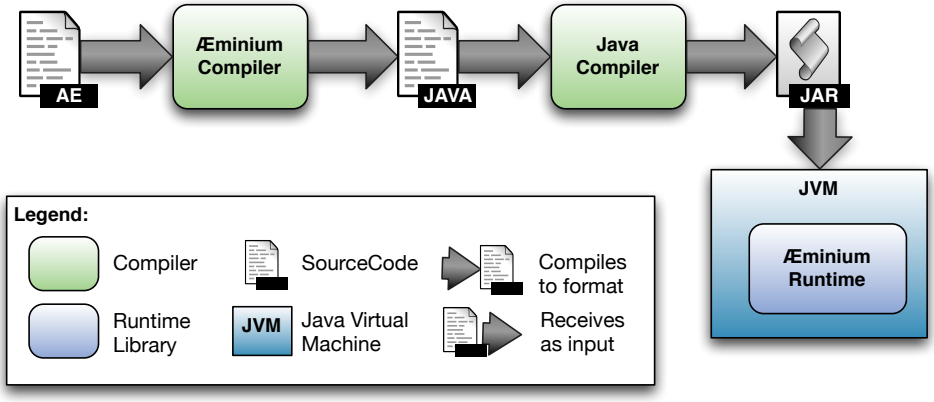

Figure 1.3: Æminium Architecture

The Æminium Compiler receives a Æminium source program and compiles it down to Java. The generated Java code contains tasks, groups of instructions. Moreover each task has a list of other tasks it depends upon.

The Java code is then compiled using a regular Java compiler (*javac*) and executed on a JVM. The program consists of calls to the Æminium Runtime library submitting a task for execution. Each task awaits until all the tasks it depends upon are done. Then, they are marked as ready-to-execute. The ÆminiumRuntime will execute simultaneously as many tasks as possible, limited by the number of CPU cores.

1.3.3 Research areas

The research being done on the Æminium project is divided in three areas:

- A compiler for the Æminium language that generates tasks and dependencies between them;
- A runtime that executes tasks on the hardware resources (such as multicore CPUs);
- Static verification of the system that ensures no race conditions are present.

Along the second line of research, new opportunities for improving the performance of programs are always being considered. Since Æminium is all about concurrency and parallelism, the GPU presents itself as a suitable platform for exploration. The proposed work intends to extend the Æminium system to target GPUs besides the already supported multicore CPUs. This thesis will thus present a solution for how that extension, named ÆminiumGPU, should be done.

1.4 Goals

ÆminiumGPU aims to explore the computational power of GPUs for General Purpose Computation (GPGPU). The main idea is to improve performance of programs written on a high-level programming language without the need to write architecture specific code.

This research area is still in its early years — not even two decades old — and the work performed until now is mostly exploratory. The caveat is in dealing with different architectures and programming models, for CPUs and GPUs respectively.

ÆminiumGPU extends the current Æminium ecosystem by:

- Introducing and defining the usage of data-parallel operations in the Æminium language;

- Improving the performance of programs that use those data-parallel operations;

- Allowing programmers to target GPUs more easily than when using low-level GPU languages.

Regarding the first contribution, the programming style in which one writes CPU programs is quite different the one for GPUs. The programming style refers to the semantics of a particular structure of code. For instance, for the same problem of counting elements of a list, there are, at least, two programming styles that can be used: counting by iteration or by recursion.

In Æminium, the underlying hardware can be completely abstracted and unknown for the developer. For instance, the programmer does not know how many cores the CPU has. Thus, it is only natural for the programmer not to be concerned whether the code will execute on the GPU or the CPU. In order to do so, the programming style should be platform independent and be able to express operations that can be executed in either architecture. Figure 1.4 illustrates an example in which, despite the main part of the program executing on the CPU, there is a block of code that can run on the GPU without requiring any extra language constructs.

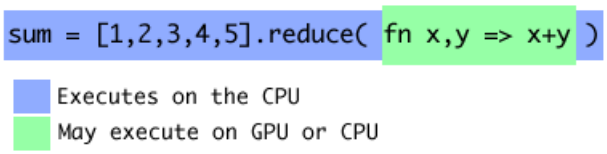

Figure 1.4: Representation of a possible abstraction of data-parallel abstractions for CPU and GPU.

Regarding the second point, data-parallel programs should not be slower than their sequential counterparts. There are certain kinds of operations that are slower on GPUs and those should execute on the CPU. But, if the performance of a program can be improved by executing on the GPU, the platform should take advantage of that.

Finally, general purpose programmers (not necessarily Æminium programmers) should be able to use this new platform to write code that makes use of GPUs. In order to achieve that, the understanding of the GPU programming

model should not be a requirement. All the details should be abstracted by the ÆminiumGPU.

The work on this thesis focuses on the conception, implementation and evaluation of the GPU extension of Æminium, leaving out any other aspect of the Æminium language.

1.5 Contributions

This work gives the following contributions:

- Introduction to the Æminium language of a programming style and primitives that allow code to target both GPUs and CPUs;
- Definition of compilation and execution plans for programs on the ÆminiumGPU;
- Implementation and publication of a framework for executing programs on GPUs, including a compiler, runtime and auxiliary libraries;
- Definition and implementation of optimizations for the execution of programs on the GPU to avoid common bottlenecks;
- Definition of a mechanism for predicting the performance of algorithms on the CPU and the GPU based on their complexity and size, and on empiric data collected prior to execution.
- Evaluation of the performance of the proposed solution.

1.6 Document Structure

This document is organized in the following way: this chapter introduces the proposed work and defines the context and its goals; Chapter 2 discusses related work; Chapter 3 introduces the approach and the followed methodology; the solution is described in more detail in Chapter 4; Chapter 5 evaluates the present solution; Chapter 6 describes future work; finally Chapter 7 draws the conclusions.

2

State-of-the-Art

This Chapter discusses the benefits and caveats of the usage of GPUs for general-purpose programming and how current programming languages integrate such features.

The first section introduces GPUs as programmable units and as targets for General Purpose Programming. The second and third sections focus on state of the art languages for the GPU, in low and high level programming, respectively. The last section introduces the Map-Reduce algorithm and its parallel implementations for different platforms.

2.1 Programming on GPUs

For many years, the single purpose of GPUs was to accelerate certain parts of the graphics pipeline. The focus of manufacturers was in games, 3D renderers and, more recently, video decoding algorithms.

Traditionally, the GPU was just an output device. Applications would generate the display data and send it to the graphics card. There was no control over what happened after that step.

DirectX 8, Microsoft's set of APIs for graphics generation, introduced programmable vertex and pixel shaders. These customizable shaders allow programmers to apply effects such as shadows, lightning, translucency and virtually any effect that only requires changes to the position and colors of vertices. Only in 2001 was that specification included in a product, NVidia's GeForce3[26].

DirectX 9, in late 2002, and OpenGL, in 2003, introduced their high-level APIs for programming vertexes, GLSLs and HLSLs respectively, both featuring a C-like syntax. These two languages are similar in their nature, but they differ in the specifics, such as type naming and matrix conventions[34].

Listing 2.1: Brook Example Code

```
1 stream Line {
2   vec2i a;
3   vec2i b;
4 }
5
6 kernel void vtransform (Vertex vtx, Vertex out tvtx, mat4f matrix)
      {
7    tvtx.pos = matrix * vtx.pos
8 }
```

Since 2001, NVIDIA has developed a more platform-agnostic solution. Cg[23] is another shading language that aims to be the "c for graphics". The Cg toolkit includes real-time processing features that map directly to the hardware. One example is binding variables to a hardware resource, such as the GPU or CPU. The memory for each variable is allocated on the respective device.

The output of the Cg compiler can be either assembly, GLSL or HLSL, making Cg a preferable choice for developers who need to support both platforms. Thus, Cg is a very popular choice among game studios, such as Blizzard, Codemasters, Eletronic Arts, id, NAMCO, Sega, Sony and Valve[25].

2.2 Low-level GPGPU languages

The programmability of the graphics pipeline allowed researchers to try and use the computational resources of the GPU for other purposes other than generating graphics. BrookGPU[10] was one of the first projects to emerge from that interest.

Brook introduces the concept of streams, a collection of data which can be operated in parallel, similar to arrays. This C-based language also features some higher-level primitives, such as reduction, scattering, gathering and search. These are operations that fit naturally in the graphics card model. Brook was also the first language to incorporate the `kernel` keyword to represent a function that executes on each GPU core. An example of a kernel is present in listing 2.1.

Later in 2003, Michael McCool[24] started to work on a meta-programming library for C++, which generates assembly code at runtime. This meta-programming technique allows programmers to tune the generated code, as well as to activate and deactivate features. Sh is claimed to be a good option

for research, since it uses an intermediate language that can be compiled to low-level assembly. Future primitives can be added to the compiler backend without any change to the programs.

In early 2007, NVidia released the first version of CUDA, a compiler for C-like kernels and an API for executing such kernels and transferring data between the main memory and the graphics card memory. Around the same time AMD was also developing a GPGPU framework called Close-To-Metal, that was evolved into Stream SDK, which uses the OpenCL standard.

Although OpenCL was initially developed by Apple, it was proposed as a standard to the Khronos Compute Working Group, already responsible for OpenGL. Intel, NVIDIA and AMD are all members of the Khronos Group and, as of the third quarter of 2010, represented together 99.1% of GPU sales[4].

CUDA and OpenCL are a big improvement over previous GPU programming languages. For a start, they were both designed for general purpose computations. Previous languages were designed for shading, but used for GPGPU because there was no other option.

These two languages expose a shared memory on the GPU, which can act as a local cache during calculations. Furthermore, both allow reading from and writing to multiple and arbitrary locations in memory (scattered read/writes). Previous languages were limited to reading data from textures without writing in the same execution. Copies from main to GPU memory are also faster in graphics cards that support these APIs.

On a syntactic level, OpenCL and CUDA have type integration between the CPU and GPU. This means that developers can define their own data structures which can be used seamless on either platform. Both languages also support functions that are defined once and can be executed on the CPU and GPU.

Since all current cards that support CUDA also support OpenCL, it is more interesting for generic applications to support more brands of hardware by using OpenCL. OpenCL has also the advantage of supporting multiple and heterogeneous platforms on the same machine. For instance, it is possible to run kernels in one x86 CPU, one ATI and another NVIDIA GPU at the same time. Some of the OpenCL implementations can be seen in Figure 2.1 in their layers.

2.2.1 OpenCL Programming Model

OpenCL is the standard language and API for performing GPGPU and it is supported by discrete cards from different manufacturers. Because GPUs ands CPUs have a different internal architecture, they should be programmed

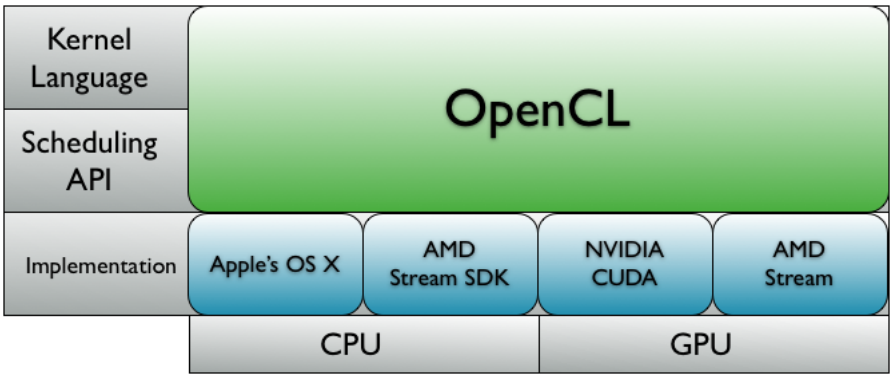

Figure 2.1: Different OpenCL implementations.

differently. Thus, in order to achieve better performance in applications, it is important to understand how OpenCL works.

OpenCL is an abstract architecture and may be mapped to CPUs or GPUs. In the scope of this project we are focusing on GPUs, since CPU parallelisation is already done at the Æminium runtime level.

OpenCL abstraction	GPU hardware
compute device	GPU (ATI Radeon HD 4870)
compute unit	SIMD
processing element	thread processor
work item	pixel, vertex
work group	group of wavefronts
local memory	local data share
private memory	scratch memory
global memory	global buffer
vector variable	one or more 128-bit registers

Table 2.1: Mapping of OpenCL concepts to a GPU

To understand how the abstract concepts of OpenCL map into the hardware level, an example for the ATI Radeon DB 4870 card in shown in Table 2.1[8]. The compute unit is a Single Instruction Multiple Data (SIMD) unit, which means that all of the cores will execute the same instruction at the same time, each with its own data. SIMD units are ideal for operations on an array that are independent across elements.

Due to the phenomenon of locality, GPUs have different memories and caches within their structure. In order to write performant OpenCL code, it is important to understand the layout on the graphics card, pictured in

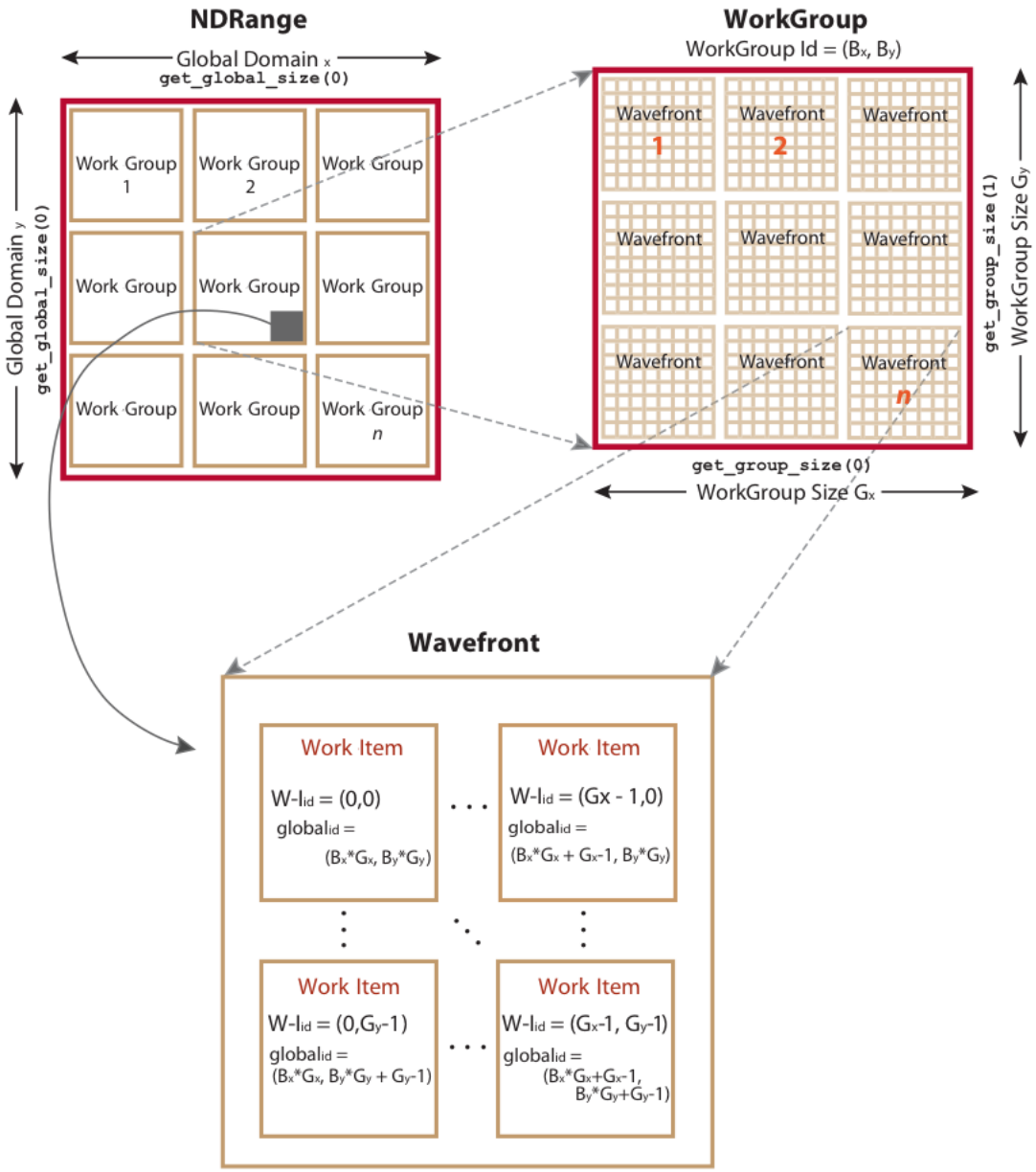

Figure 2.2: Work Item grouping according to the GPU hardware.

Figure 2.2. In OpenCL, the **NDRange** is the largest unit to which one may send work to. It depends on the nature of the problem to solve (and more directly on the kernel itself) and it could be 1, 2 or N dimensional.

For instance, when summing two vectors, the best NDRange choice would be 1D. This way, each work item handles the same index of the input and output vectors. When doing matrix multiplications, a 2D range is preferable since each work item receives different indices of both matrices as input.

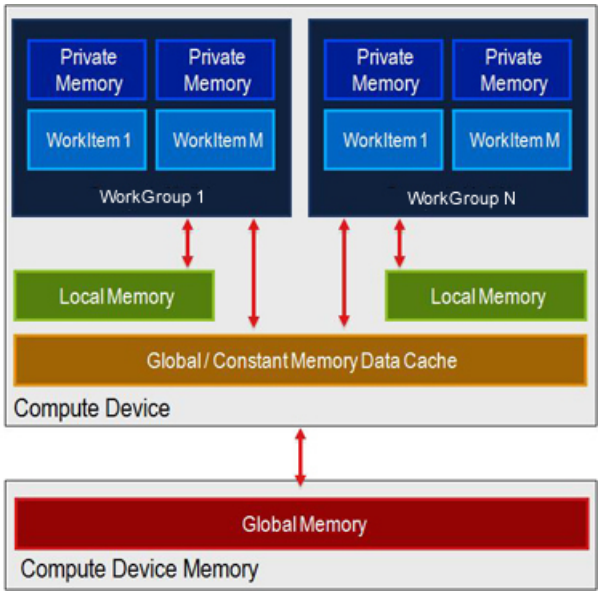

Figure 2.3: Memory Architecture for OpenCL on GPUs.

Inside each computing SIMD, work items are arranged on a square structure of workgroups, called wavefronts by AMD or thread blocks by NVIDIA. This organization is important since workgroups may have a shared memory that might improve performance. Therefore, it is beneficial to organize the data and the NDRange so that work items next to each other use adjacent data from input buffers.

As for workgroup size, one should schedule a multiple of 32 up to 512 to fit the hardware layout of most graphics cards. Each controller can execute up to 8 groups concurrently. Thus, the size of the groups should not be too big to divide into concurrent executions, nor so small that it prevents achieving a good occupancy. Although there are profilers that measure the occupancy, the best workgroup size is usually obtained empirically.

An overview of the different memories can be seen in Figure 2.3. Each work item has its own private memory, which are mostly registers, but shares a local memory across workgroups. Local memory has a latency of just a few

cycles and a throughput of up to 44GB/s per multiprocessor, over 1.4 TB/s per GPU. The global memory of the GPU, the one that synchronizes with the host memory, has a latency between 400 and 800 cycles and a throughput of 140Gb, for a 1GB board. Given this values, it is beneficial to use the local memory as much as possible.

Global data in the OpenCL model is not coherent since different workgroups execute independently and order is not guaranteed. Execution can be explicitly synchronized by using OpenCL atomic primitives, such as *Barriers*.

Divergence is an inherent effect to this architecture. Any control flow instruction (if, switch, do, for, while) can significantly affect the throughput of the kernel if the work items diverge. Operations inside a workgroup execute at the same time (GPUs are SIMD - single instruction for multiple data). If there are different control flows, only one operation can be executed at a given time thus serializing the different branches.

Given an example kernel that does one operation for odd and other for even indices, the execution inside the workgroup would take twice the time. Since workgroups execute independently, a better solution would be to reorder the array so odd indices would be aligned with half of the workgroups and even indices with the other half. The corresponding example code can be seen in Listing 2.2.

Listing 2.2: OpenCL example with and without divergence

```
1 // Example with divergence
2 if (threadIdx.x % 2 == 0)
3     // something
4 else
5     // different something
6
7 // Example without divergence
8 if (threadIdx.x/WORKGROUP_SIZE % 2 == 0)
9     // something
10 else
11     // different something
```

2.3 High-level GPGPU languages

Upon the introduction of both CUDA and OpenCL, programmers felt the need to integrate the power of the GPU with high-level languages and platforms such as Java, .NET, Python, Ruby and others. Bindings such as JavaCL

and PyCuda exposed the C API functions to those languages, but still required the programmer to understand all the details and schedule executions manually. Also the kernel code would always had to be written in OpenCL. More recently, several high-level programming frameworks are being developed such as the ones detailed below. These framework are abstractions over CUDA or OpenCL, still using one of those languages as output.

2.3.1 JavaCL

JavaCL[1] allows the programmer to call the OpenCL C API within Java. The main difference to the C API is the need to use Buffer objects to send arrays to the GPU back and forth.

Similarly to other bindings, JavaCL is as low level as the C API. It just makes OpenCL operations possible in a high-level language. The limitations are the same:

- Kernels need to be written in OpenCL code;
- Memory must be copied explicitly;
- Execution parameters, such as workgroup size and NDRange, have to be specified.

On the other hand, these limitations might also be an advantage in situations when programmers need to fine tune their applications. It is possible to improve the performance of applications by changing some parameters in low-level calls to the OpenCL API.

JavaCL and other language bindings are relevant since they are the foundation for higher-level projects.

2.3.2 ScalaCL

ScalaCL[2] is a project based based on the JavaCL bindings that aimed to make the usage of GPU programming easier in Scala. The first version implemented a Domain-Specific Language for GPGPU by overriding methods on default types.

Methods on native types, such as `+` on `Int`s, are not evaluated strictly, but rather represented as a lazy operation. If a sequence of those methods are called, the AST-like representation of operations is compiled to OpenCL and executed on the GPU. All these steps happen at compiler-time.

Work on a second version has since been started, and it is still ongoing. This second version introduces a collection library and a compiler plugin. The

collections library has the same API as the standard Scala collections, but implemented in a way that methods of arrays can be executed on the GPU using the JavaCL bindings. The compiler plugins selects the functions passed as arguments to those methods and converts them to OpenCL. This conversion is limited to a subset of the language.

2.3.3 Atomic HedgeHog

Atomic HedgeHog(AHH) [28] came from a neurobiology project that had the need to use GPUs to perform several scientific calculations. The resulting library exposes decorators to explicitly execute a function on the GPU. These decorators are wrappers around regular Python functions that handle the conversion to OpenCL code. As an example, the code for performing the sum of two vectors is shown in Listing 2.3. Notice that the arguments of the *sum* function have no type, making that function work as a template for all concretizations for specific types. However there is still some boilerplate for GPU-specifics, such as the context, in and out arguments and the global size.

Listing 2.3: Vector Sum in Python using Atomic HedgeHog

```
1 @cl.oquence.fn
2 def sum(a, b, c):
3     gid = get_global_id(0)
4     c[gid] = a[gid] + b[gid]
5
6 ctx = cl.ctx = cl.Context.for_device(0, 0)
7 a = numpy.random.rand(50000).astype(numpy.float32)
8 b = numpy.random.rand(50000).astype(numpy.float32)
9 c = numpy.empty_like(a)
10
11 sum(ctx.In(a), ctx.In(b), ctx.Out(c), global_size=a.shape)
12 print la.norm(c - (a+b)) # should be ~0
```

AHH reuses the Python compiler to generate the AST from the sum function and compiles it down to OpenCL. The execution on the GPU is still explicit and the kernel function should be programmed using OpenCL functions.

AHH is a good improvement over simple bindings by supporting kernel programming in the Python language that might reuse regular Python functions. AHH also integrates closely with SciPy scientific library.

Listing 2.4: Dot Product Example in Haskell with Accelerate

```
1 dotp :: Vector Float -> Vector Float -> Acc (Scalar Float)
2 dotp xs ys = let xs = use xs
3              ys = use ys
4           in fold (+) 0 (zipWith (*) xs ys)
```

2.3.4 Accelerate - Haskell CUDA Backend

A common scenario in Functional Programming is the application of the same operation over each element of large data structures. That pattern is defined as the ubiquitous *map* operation and its variants.

A group of researchers from the University of New South Wales in conjunction with NVIDIA developed a CUDA-powered backend for Haskell named Accelerate[12]. Their project allows specific Haskell code to be executed in a secondary backend, in which certain scalar operations are translated to CUDA and scheduled on the GPU at runtime.

Accelerate is based on the concept of skeletons. A skeleton is an OpenCL code template for a higher order function such as *map*, *fold* and *zipWith*. These functions represent patterns around an argument function that handles the specifics of the operation.

Accelerate overloads these functions on the types that have a CUDA match. The argument function is converted to a CUDA kernel and executes on the backend. Results are returned back in Haskell types. The whole process is almost invisible to the user.

Listing 2.4 shows the Dot Product function written using Accelerate. It is very similar to a pure CPU version, the only differences being the `Acc` monad on the return type and the usage of the `use` constructor, for allowing lists to be copied to the GPU.

2.3.5 Another Parallel API - Aparapi

In October 2010, AMD released a pure Java API called Another PARallel API. Aparapi is a Java library that allows the programmer to extend the `Kernel` class, and to write Java code inside it. A simple example is shown in Listing 2.5. By calling `Kernel.execute()` the Aparapi library will convert the Java bytecode to OpenCL and execute it on the first GPU found.

The project is still in the alpha version and it has as a few limitations:

- Aparapi does not support external method calls inside the kernel, which is something that limits its extensibility;

- Aparapi only supports native types;
- Only AMD GPUs are supported at the moment;
- Only one GPU may be used.

On the other hand, Aparapi supports calls from multiple threads and provides an interesting fallback mechanism. If no GPU is found, Aparapi uses a thread pool to split the work across the host multi cores. If only one CPU core is available, code will be executed sequentially.

Listing 2.5: Aparapi simple example

```
1 class SquareKernel extends Kernel{
2     private int values[];
3     private int squares[];
4     public SquareKernel(int values[]){
5        this.values = values;
6        squares = new int[values.length];
7     }
8     public void run() {
9         int gid = getGlobalID();
10        squares[gid] = values[gid]*values[gid];
11    }
12    public int[] getSquares(){
13        return(squares);
14    }
15 }
```

2.4 Map-Reduce

2.4.1 Map and Reduce Functions

Map is a higher-order function present in many programming languages. In a functional form it is called *apply-to-all*. The function takes a list of elements and a function, usually called lambda function. Map returns a list of applications of that function to each element of the list. Mathematically, map can be expressed by Equation 2.1. Figure 2.4 shows a visual representation of $map(x- > x * x, [1, 2, 3, 4])$, which would return $[1, 4, 9, 16]$.

(2.1) $map(f, [a_1, a_2...a_n]) = [f(a_1), f(a_2), ..., f(a_n)]$

The mathematical nature of the map function allows for a few optimizations. Being f and g two functions and xs a list, $(map(f) \circ map(g))(xs)$ is the

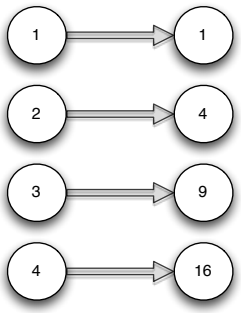

Figure 2.4: Example of map operation with the square as its lambda.

same as $map(f \circ g)(xs)$. This is called *map-fusion* and allows for two functions to be applied at once inside a map. Another optimization, explored later on, is that each element of the output is evaluated independent of the order. Thus, it is possible to calculate each value in parallel.

Reduce is another higher-order function, also known as *fold*, *accumulate*, *compress* or *inject*. Reduce converts a list of elements into a smaller one, usually with a single element. Reduce works by iterating an arbitrary function, again henceforth referred to as reduce lambda, over a list of elements, building up a return value. Reduce can be mathematically expressed as Equation 2.2. Furthermore, a visual representation of *reduce* is pictured in Figure 2.5.

$$result = foldl(f, a, seed) = f(f(f(seed, a_1), a_2), ..., an) \tag{2.2}$$

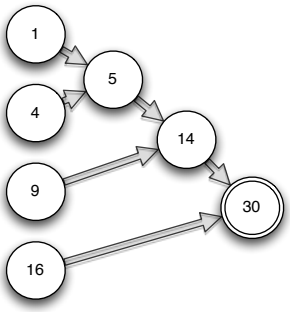

Figure 2.5: Example of a reduce operation, with sum as the reduction lambda.

There are two flavors for reduce, a left variant and a right variant, depending on which direction elements are combined. The right reduce starts with the first element while the left starts with the last one and goes on to the first.

In different languages and compilers this might yield in better performance due to stack allocation on what can be recursive call.

Most implementations allow for a third parameter: the seed. The seed is the initial value of the accumulator that stores the combining results. It is most important when the type of the result is different from the type of elements in the list. Take the example of reducing a list of elements to a tuple in which the first element is the sum of odd values while the second is the sum of even values. The seed would be the tuple $(0, 0)$ and the reduce lambda would take as arguments the previous tuple and the current element. Hence, it is possible to implement a type safe reduction between different types.

The iterative nature of reduce makes it uninteresting for the purpose of parallel programming. However, given the assumption that the reduce lambda is an associative and commutative function, the order in which aggregation happens does not matter anymore. With this constraint, it is now possible to re-organize how elements are combined together in a tree-like order. Figure 2.6 shows the same example as before, now using the tree structure. In this ordering, elements are aggregated two by two at the same time, and then the same happens to intermediate results until only one element remains.

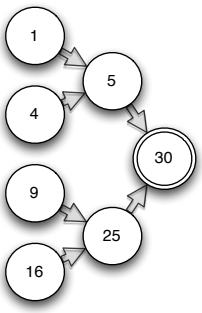

Figure 2.6: Same example of a reduction, but done in parallel with one less level.

By applying this change in the algorithm, it is possible to run operations in parallel and reduce the number of levels from N to $log_2(N)$.

Map and **Reduce** are commonly used together, as seen in Figure 2.7. In this sequence, operations can be merged together, in a *map-fold fusion*. Immediately before the aggregation, elements are replaced by the application of the map lambda. Given a reduce lambda f, a map lambda g and a list xs,

Equation 2.3 applies.

$$reduce(f, map(g, xs)) = reduce(x, y- > f(x, g(y)), xs) \quad (2.3)$$

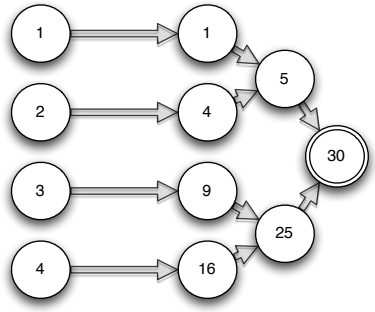

Figure 2.7: Example of map followed by a reduce.

2.4.2 MapReduce Frameworks on Clusters and Multicore

Map-Reduce is a programming pattern derived from functional programming and popularized by Google[13] for high performance data analysis. This technique is used because of its parallel nature, which allows it to be scaled to several machines, and even clusters. Besides the implementation from Google, the open-source framework Hadoop[9] has become quite popular for analysis and processing of large data.

This model distributes work over different machines. Figure 2.8 shows the pipeline of the framework. The user programs two different operations, one for mapping and another one for reducing. Each one may be replicated over several machines. The input data is split into different chunks, and each chunk is tagged with an ID. From then on, data will be key-value based. Each worker will take a chunk, process it, and deliver it to the respective reduce worker. In the end, reduce workers produce several outputs, which are combined in a post-operation merge. There is an implicit sort between maps and reduces, in which all values for a key are aggregated together for a certain worker.

Phoenix[35] is an implementation of Google's Map Reduce for shared memory multicore systems. An evaluation of Phoenix[31] has shown similar performance to the optimized pthreads version for most applications. However, for certain cases that do not fit the MapReduce model, the overhead is quite significant.

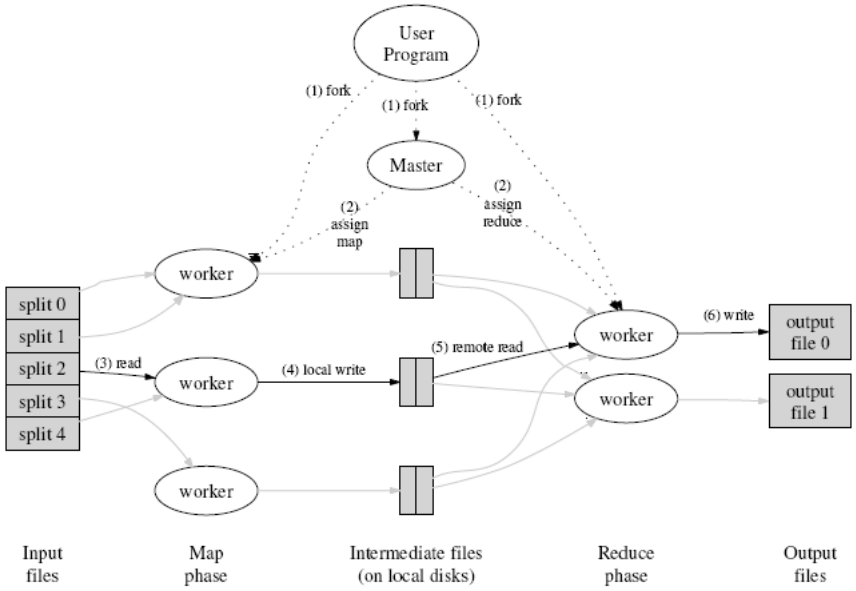

Figure 2.8: MapReduce pipeline in Google MapReduce framework[13].

2.4.3 MapReduce Frameworks on GPUs

Mars[17] brought the idea of MapReduce frameworks to use GPUs. Due to the limitations of the platform, several changes in the architecture were made. For instance, since there is no dynamic memory allocation on the GPU, all the memory is reserved before the kernel even launches. Since the result after the reduce stage is unknown, more than necessary memory is allocated beforehand.

A write conflict occurs when multiple threads try to write to the same shared output array. Thus, Mars has two stages: one for map and other for reduce. The former avoids collisions due to the deterministic nature of map. Output positions will be always fixed and non-deterministic. The latter uses the prefix sum technique[16] to accommodate the results sequentially in the output array.

Another optimization for key-value map reduce is to use vector types. This means that group of key and values offsets and sizes in the corresponding arrays can be stored as a `int4` instead of several arrays of `ints`. Recent GPUs support these vector types which fetch the vector in a single memory request, improving the performance of the algorithm.

2.4.4 Reduce Implementation Details on GPU

One of the operations of the Map-Reduce pattern is reduction, an aggregation of several values into only one. It is considered a common and important data

parallel primitive that is part of more complex algorithms.

The standard definition of a reduction is sequential and linear. It is possible to implement a parallel version, assuming that the outcome of the operation is not influenced by the order in which elements are aggregated together.

With that assumption it is possible to perform the reduction in passes. In each pass the input is split in groups of elements, and each group is reduced in parallel with the others. Figure 2.9 shows the different passes as horizontal V shaped lines and the parallelism inside each one.

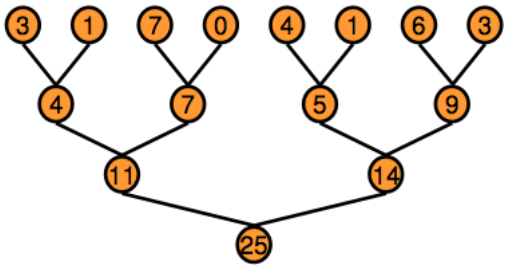

Figure 2.9: Tree based approach in parallel reduction[15].

The three major OpenCL vendors, NVIDIA[15], AMD[11] and Apple[6], all explore the implementation of this reduction in OpenCL. All three implementations are very similar, only being distinguished by the specifics for each platform. NVIDIA optimizes for CUDA cards, for instance.

In the cited white papers and examples, the reduction is optimized in steps. Each step overcomes a limitation specific to GPUs by taking advantage of its architecture.

The first step when designing the algorithm is the lack of global synchronization between all workers. This makes it impossible for a kernel to reduce a large array at once. The solution to overcome this limitation is to use the kernel invocation from the CPU as a synchronization barrier across all work units. Despite being called different times, the kernel code itself is the same, thus saving compilation time.

Another optimization is the usage of the shared memory of the GPU for intermediate results while reducing inside each worker. Only the final result is sent back to the GPU for the next pass. The performance improves because reading from and writing to shared memory is faster than using the global memory.

Listing 2.6: Loop unrolling on reduction kernel

```
1 for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
2   if (tid < s)
3     sdata[tid] += sdata[tid + s];
4   __syncthreads();
5 }
6
7 if (tid < 32) {
8   sdata[tid] += sdata[tid + 32];
9   sdata[tid] += sdata[tid + 16];
10  sdata[tid] += sdata[tid + 8];
11  sdata[tid] += sdata[tid + 4];
12  sdata[tid] += sdata[tid + 2];
13  sdata[tid] += sdata[tid + 1];
14 }
```

Reducing thread divergence is another issue when writing the code for the algorithm. All threads should be executing code at all times. Another optimization is to unroll the last iterations of an inner loop, as seen in Listing 2.6. For instance, if each workgroup has 32 threads, the last 6 iterations can be unrolled since the workers inside that workgroup are all synchronized. Those iterations will not have the overhead of the synchronization barrier calls.

3 Approach

This chapter describes the strategy followed in the extension of Æminium for GPUs. The first section presents the ÆminiumGPU programming style, this is, how developers interact with the platform. Then, the general architecture is described and the main components are detailed. The third section explains how this architecture allows other developers to extend operations and structures. The last section explains the development methodology and practices for the development of the solution.

3.1 Programming Style

The Æminium language has both object-oriented and functional programming roots. The proposed style comes from the latter and makes heavy use of higher order functions.

As shown in Section 2.4, the Map-Reduce pattern is broadly used in programs to distribute data-parallel operations over different programming units (from GPU and CPU cores to machines across a network). In fact, higher-order functions are a good fit for the programming style of ÆminiumGPU due to the fact that they can abstract parallel operations similarly to templates, or skeletons. In their nature, higher-order functions separate programming patterns from user-defined functions. The implementation of the former can be done either sequentially on the CPU, or in parallel on the GPU, as long as it supports the user-defined function.

Although it is inspired by FP, Æminium is, in its core, a Object-Oriented language. As such, the programming API should also be object-oriented. The selected approach is based on Scala Generic Parallel Collections(SGPC) [30] and it is object-oriented. SGPC was designed for multicore parallelism. This approach is based on typed lists, in which elements all have the same

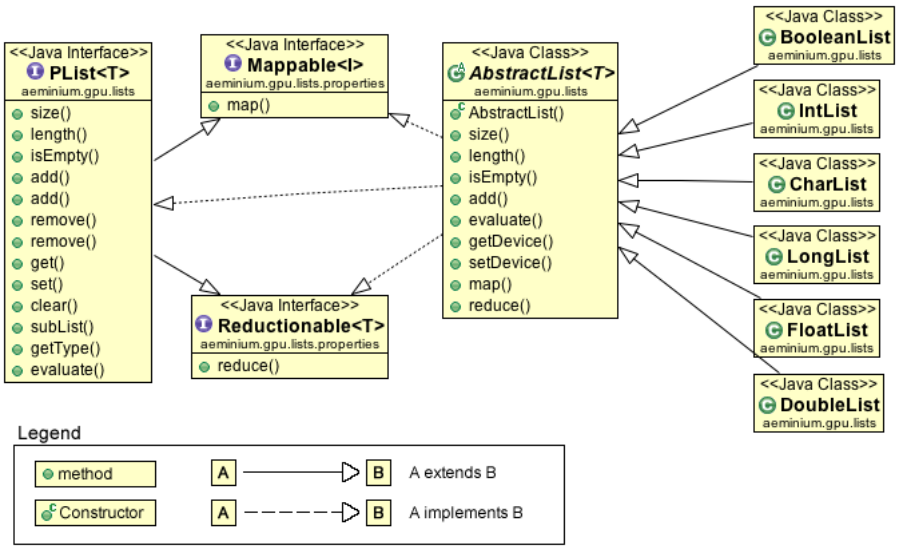

Figure 3.1: Class Diagram of Æminium GPU Collection. Methods are not represented by their full signature.

type. ÆminiumGPU provides the programmer with `IntList`, `FloatList`, `DoubleList`, `CharList`, `BooleanList` and `LongList` as containers for several elements of `ints`, `floats`, `doubles`, `chars`, `booleans` and `longs`. Figure 3.1 shows their common superclass `AbstractClass` and the interfaces that they implement.

The implementation of these PLists is done very similarly to the standard ArrayList in the Java library. The major difference is the two extra methods defined by the interfaces `Mappable` and `Reductionable`, `map` and `reduce`, shown in Listing 3.1. This way, the functional map and reduce are available for the developer in an object-oriented fashion as methods of lists.

Listing 3.1: PList public methods that expose primitives

```
1 public PList<O> map(Lambda<I,O> mapLambda);
2 public PList<I> reduce(ReduceLambda<T> reduceLambda);
```

The methods in Listing 3.1 allow programmers to write map and reduce sequences such as the one of Listing 3.2. This small Java snippet represents the sum of the square numbers of the input array. The verbosity in this code is due to the lack of first-class lambdas in Java, which were written as anonymous inner classes. The same program in Æminium would be more readable, shown in Listing 3.3.

Listing 3.2: Example of summing the square of the elements in one array using ÆminiumGPU map and reduce in Java.

```
1 Integer SumOfSquares = input.map(new LambdaMapper<Integer,
      Integer() {
2   @Override
3   public Integer map(Integer input) {
4     return input * input;
5   }
6 }).reduce(new LambdaReducer<Integer>(){
7   @Override
8   public Integer combine(Integer input, Integer other) {
9     return input + other;
10  }
11
12  @Override
13  public Integer getSeed() {
14    return 0;
15  }
16 });
```

Listing 3.3: Example of summing the square of the elements in one array using ÆminiumGPU map and reduce in Æminium.

```
1 sum_of_squares = input.map(fn (i) => i*i).reduce(fn (i,o) => i+o
      , 0);
```

3.2 Architecture

As mentioned in Section 1.3, the Æminium compiler is still in development. Thus, it is not yet stable and does not implement the full language specification. This constrains the integration of ÆminiumGPU with the compiler.

To cope with the lack of a compiler, it was chosen to use Java as the source language. Java was chosen because it is the target language of the Æminium Compiler and because the Æminium Runtime already supports it.

The Æminium system is divided in two components: the compiler and the runtime. For ÆminiumGPU it was decided to increment each one, by adding a second compiler and a second runtime library.

The proposed architecture is pictured in Figure 3.2 and introduces two new elements: ÆminiumGPU Compiler and ÆminiumGPU Runtime. For the execution of an Æminium source code, the following steps are taken:

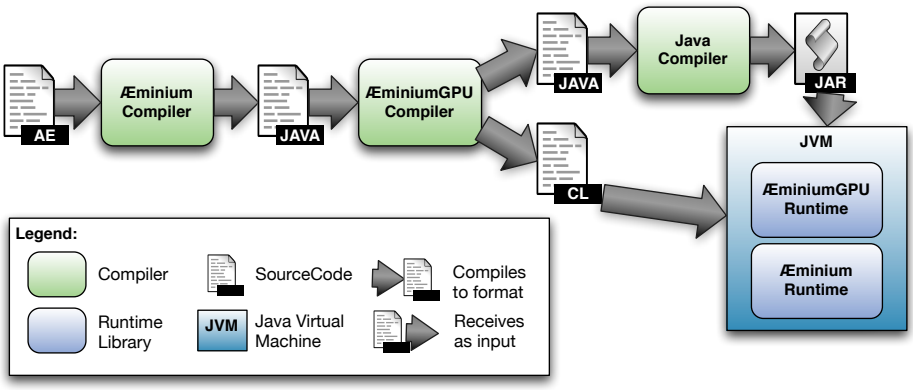

Figure 3.2: Pipeline for ÆminiumGPU

- The input Æminium source file is fed to the Æminium Compiler;
- The resulting Java code is passed on to the ÆminiumGPU compiler. In this step, the ÆminiumGPU compiler generates OpenCL code and modifies the Java file to call OpenCL kernels;
- The new Java output is compiled by a Java Compiler into bytecode. The bytecode can be packaged in either class files or a single JAR.
- The Java bytecode is executed on the JVM. Execution requires both the Æminium and ÆminiumGPU runtimes. The Æminium Runtime is responsible for parallelizing different tasks on the CPU multiple cores and for parallelizing data-oriented operations on the GPU.

3.2.1 ÆminiumGPU Compiler

The ÆminiumGPU Compiler identifies the code that is able to run on the GPU, translates it to OpenCL and modifies the original code to use that OpenCL code.

The ÆminiumGPU Compiler performs the following steps:

- Parse the input source code and generate an Abstract Syntax Tree(AST);
- Transverse the AST and identify lambdas (either for *map* or *reduce*);
- Translate lambdas to OpenCL. If translation is not possible, that lambda is left unmodified.
- Append methods to the lambdas, returning the OpenCL code as string;

- Compile kernels to the GPU.

These steps will be described in more detail in Section 4.1.

It is important to notice that not all Java code can be translated to OpenCL. The ÆminiumGPU compiler does not support method calls, non-local variables, for-each loops and object instantiation. It does support:

- Arithmetic and Logical instructions (+,−,*,<=, ...)
- The `return` statement
- Native types (`int`, `boolean`, `float`, `double`, `char`, ...)
- Arrays of native types(`int[]`, `boolean[]`, ...)
- `if`, `for` and `while` constructs
- `break` and `continue` statements
- The `switch` construct for native types
- Local variables definition and access
- Static methods on the `java.util.Math` class

Whenever the lambda operation includes a non-supported feature of Java, compilation for that lambda fails. The compiler outputs the exact input code without any modification. This will trigger the runtime to execute the code on the CPU by default.

3.2.2 ÆminiumGPU Runtime

The ÆminiumGPU Runtime is a Java library responsible for providing Æminium Programs with the data structures (and methods) described earlier in Section 3.1. It is also responsible for the execution on the GPU of the OpenCL previously generated by the Æminium Compiler.

Figure 3.3 shows an overview of the core components included in the Runtime. For the sake of brevity, several other peripheral packages and classes were omitted from the figure.

The collections package is probably the most interesting one for developers. It contains lists, and matrices, that can be used to store data. These collections have methods, namely `map` and `reduce`, that instantiate the different operations:. The `Map` or `Reduce` classes contain all the logic for executing code on the GPU. But, the generation of OpenCL Kernel code is performed separately by the `Generator` package.

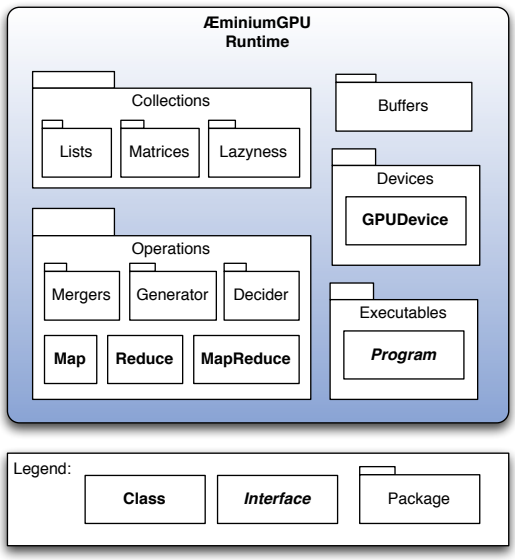

Figure 3.3: Static perspective on ÆminiumGPU Runtime.

Classes in the Operations package implement the `Program` interface, which defines an API that exposes the steps required for executing a program on the GPU. That interface is latter called from the `GPUDevice` class, which wraps the OpenCL API for sending data and code to the GPU.

The `Buffers` package includes utilities to convert data from Java to OpenCL types and copy it to the GPU.

Finally, packages `Lazyness` and `Mergers` together with the `MapReduce` class are used on the optimizations described later in Section 4.3. The `Decider` package is also used for a different optimization, described in Section 4.4.

Figure 3.4 presents a class diagram with the main classes involved in a map operation over an `IntList`. Just like other lists, `IntList` implements the `PList` interface and the methods `map` and `reduce`.

All `PLists` include a default `GPUDevice`, which is the first GPU available on the local machine. ÆminiumGPU supports more than one GPU at the same time by replacing the default with another instance of `GPUDevice`. All map-reduce operations on that specific `PList` will be performed preferably on that specific `GPUDevice`.

Figure 3.5 refers to the sequence in which the classes identified in the previous example interact. The user program calls the `map()` method on an `IntList`, which instantiates the `Map` class with that `PList`.

When requesting the output of the `Map` operation, the operation is evaluated, either on the CPU or GPU[1]. The `Map` operation calls the `execute`

[1]Normally operations execute on the GPU. However if one is not available on the machine,

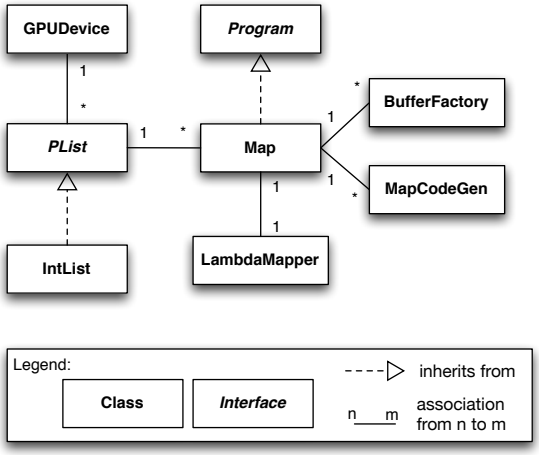

Figure 3.4: Class diagram related to the usage of a map operation over an IntList.

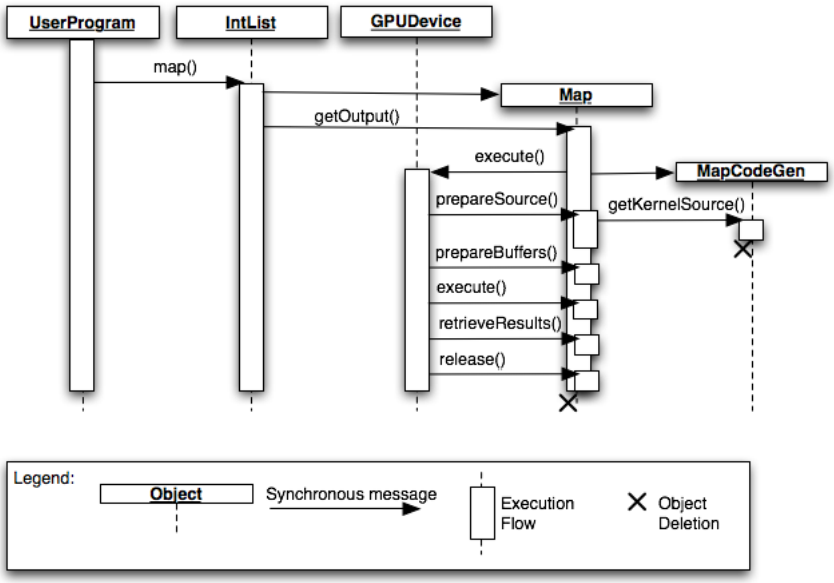

Figure 3.5: Sequence diagram related to the usage of a Map operation over an IntList.

method on `GPUDevice` passing itself as the argument. This is possible because `Map` implements the `Program` interface, an API for GPU programs. The next sequence of calls from the `GPUDevice` to `Map` are common to all operations and represents the pipeline required to execute something on the GPU:

- The `prepareSource()` method generates the OpenCL kernel code. The application of a skeleton template for all maps to this specific map is done by the `MapCodeGen` class. This step will be later detailed in Section 4.2.
- The `prepareBuffers()` method copies the data from the host memory to the GPU memory and does the required type conversions.
- The `execute()` method starts the execution of the kernel on the GPU and returns immediately. This includes the calculation of the number of workgroup and workitems per group based on the size of the `PList`.
- The `retrieveResults()` method waits for execution to complete. Then it copies data from the GPU to the GPU and converts it to Java types.
- The `release()` method manually releases all resources on the GPU and forces garbage collection on the JVM. This is required to clean the state for the next operations and prevent *Out of Memory* errors.

Although this example only calls a map operation, the execution of the reduce operation is very similar and implies the same steps.

3.3 Extensibility

Although the current implementation is limited to map and reduce operations, the same approach can be used for several other operations and structures. All higher-order functions can be expressed using the same programming style. If an higher-order function uses a collection of elements and can be parallelized, then it can be implemented in ÆminiumGPU.

In order to extend ÆminiumGPU with more data structures — such as trees and matrices — it there are two requirements. Firstly, the data structure should be represented as arrays in Java in order to be copied to the GPU. Secondly, it needs to implement the convenience methods for insertion, deletion

the operation is executed sequentially on the CPU.

and retrieval. If both occur, ÆminiumGPU provides the `map` and `reduce` methods automatically.

It might be the case that lists need more than one array. Lists of pairs is one example in which a storage in two arrays would be the best approach, even if both have the same type. In order to have coalesced accesses to the GPU global memory, indices should be consecutive across threads. By having two arrays, indices are always consecutive, at least in map and reduce. Note that ideally vectorized types such as `int2` would be a better solution. However the bindings for Java do not support those types.

Not only data structures can be added, but it is also possible to add new operations over data. New operations must implement the `Program` interface in order to comply with the ÆminiumGPU API.

ÆminiumGPU provides helpers for most of the tasks required in this kind of extension. Such helpers include: buffer helpers, which take care of copy of data to and from the GPU; and template helpers, that reduce the work involved in making generic OpenCL kernels.

In conclusion, this approach allows the straight-forwarded introduction of new data structures and operations. However, given that higher-order functions are generic and commonly used for particular applications, extending the ÆminiumGPU should not be a common task for programmers.

3.4 Planning and Methodology

The development of this project was done over the course of two semesters. The first semester was done in part-time and the second in full-time. As seen in the Gantt diagram in Figure 3.6, the first semester was focused on the assessment of the state of the art and developing the ÆminiumGPU Runtime for a preliminary evaluation. On the second semester, the focus was on writing the ÆminiumGPU Compiler, optimizing the system and performing its evaluation.

The student was located at the Centre for Informatics and Systems of the University of Coimbra (CISUC), for the first semester, and at the Institute for Software Research at Carnegie Mellon under the guidance of Professor Jonathan Aldrich, between February and May. The remaining time was completed back at the University of Coimbra.

Regarding the development methodology, work was divided in weekly sprints. During each sprint there was a meeting with the advisors. Firstly, the work done during the previous week was discussed. Secondly, the results from that work were analyzed. And finally, the tasks for the next week were defined.

The exploratory nature of this work justified a more short-time task plan-

ning. However, the overall work plan was not significantly different from the work done. The times allocated were followed, only some tasks were replaced. For instance, instead of improving the implementation of a 2D data structure such as matrix, which would be trivial, the allocated time was spent on other optimizations such as merging maps and reduces.

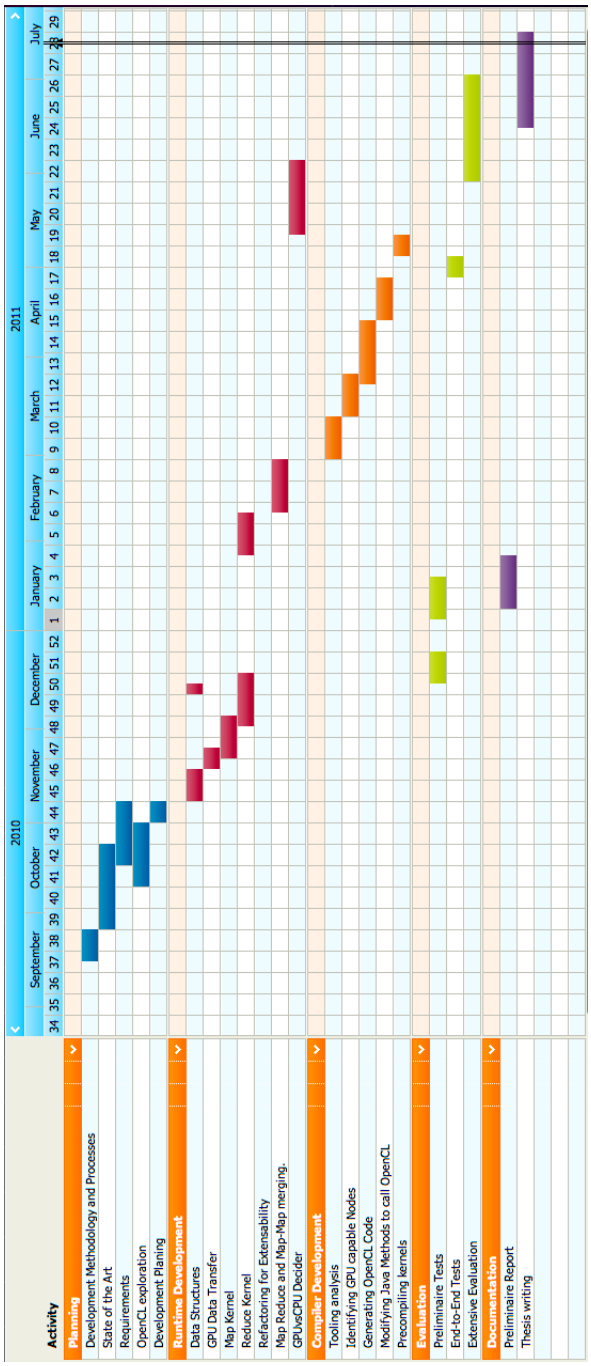

Figure 3.6: Gantt diagram of the full project.

4
Implementation

This chapter discusses the implementation of ÆminiumGPU. The first section describes the ÆminiumGPU Compiler and how it works internally. The second section explains how the runtime takes the output of the compiler and generates OpenCL Kernels. The third section explains an optimization that merges maps with maps and reduces. Finally the last section describes how the ÆminiumGPU runtime decides if a certain operation will be executed on the GPU or CPU.

4.1 ÆminiumGPU Compiler to OpenCL

The ÆminiumGPU Compiler is a compiler from Java to Java, in which the final Java code has some extra OpenCL code. This OpenCL code is the result of the translation of lambda methods. These translations will later be used to generate kernels at runtime.

To avoid creating a compiler from the ground up, and since many tools exist on the market, some of these tools were analyzed to verify if they could be used in this project. The options considered are listed in Table4.1. The comparison criteria was mainly the level of Java features supported, represented in column *Java Version*. Since these tools had different architectures and purposes, additional remarks were noted under the *Comments* column. Results are sorted by ascending supported version of Java.

Given this analysis, Spoon[29] was the tool with best ranking and best suited the type of work. Although JavaC supported version 1.6, it did not support modification of existing code. That modification is necessary since the interest in the compiler is in modifying small parts of a larger program.

The requirements for the compiler were described in Subsection 3.2.1. The

Compiler Project	Java Version	Comments
Jumbo	1.4ish	Does not support inner classes
JaCo	1.4	Does not support generics
Polyglot v2	1.4.	Too complex, monolithic.
Polyglot v1	1.5 (extension)	Deprecated.
JastAdd	1.5 (extension)	Based on Aspects.
JavaC	1.6	Can generate new code based on annotations, but cannot modify code
Spoon	1.6	Reuses JavaC and supports templates

Table 4.1: Comparison of Java2Java tools

Compiler was implemented using two passes, one for map operations and other for reduce operations. To extend the framework with additional operations, similar passes must be added.

Each pass looks for methods with a special signature. The arguments for these methods, a `LambdaMapper` or a `LambdaReducer` are then analyzed following a visitor pattern. The visitor tries to compile Java code to OpenCL as soon as it descends the AST.

Not all of the AST nodes can be translated to OpenCL. Subsection 3.2.1 lists the supported and unsupported operations. If any of the latter are found, the conversion is aborted and the final code will be the same as the input. If all instructions can be translated, the generated OpenCL code is inserted in the lambda object using the templates provided by Spoon.

Regarding the generation of OpenCL code, most of the code generation process is similar to the generation of Java code. But, there is a big difference when using native functions, such as mathematical functions and constants. Most scientific applications make intensive use of functions such as *sin*, *power*, and many others. In order to make the programming style as easy for Java/Æminium programmers, they can write code using the native Math object. All the methods and fields that have an OpenCL counterpart are translated. Unfortunately, not all type signatures match, and some casting is necessary. Although results may vary from the GPU to the CPU, the version on the GPU has always more precision by default, since type casting is conservative.

Overall, Spoon handles the tokenizer, parser (by reusing *javac*) and the generation of Java code. The ÆminiumGPU Compiler focuses on rewriting the AST to add the OpenCL version of the lambda objects. The ÆminiumGPU Compiler contains a helper to convert all possible nodes that can be used for future operations.

Finally, there is a last step on the compiler. All the map and reduce kernels

are pre-compiled, in order to reduce runtime overhead of compiling kernels. This step only detects single map and reduces operations because the chaining of these operations may depend on external factors (such as `if` statements). As such, that detection can only be done at runtime.

4.2 Map and Reduce skeletons

The two data-parallel operations currently implemented on ÆminiumGPU are map and reduce. These are implemented as skeletons, templates of OpenCL code.

Listing 4.1 shows a simplified version of the generic map kernel template included in ÆminiumGPU. Instances of `{{var}}` are replaced by the Runtime with the *var* variable. This example starts by defining the lambda function for converting an element of the input into another. The function is defined as inline to avoid the instruction overhead of calling a function and avoiding increasing the stack. The types, name of the functions and parameter names are all defined by the Runtime, thus this kernel template is generic. The *source* variable contains the OpenCL source code generated by the ÆminiumGPU compiler.

Listing 4.1: Simplified map kernel template

```
1 // Specific Map Lambda Function
2 inline {{output_type}} {{map_lambda_name}}({{input_type}} {{
      map_lambda_par}}) {
3   {{source}}
4 }
5
6 __kernel void {{map_kernel_name}}(__global const {{input_type}}*
      map_input, __global {{output_type}}* map_output) {
7   int map_global_id = get_global_id(0);
8   map_output[map_global_id] = {{map_lambda_name}}(map_input[
        map_global_id]);
9 }
```

The map kernel itself is straightforward. It receives two arrays, a read-only array and a write-only array. The `const` keyword is important to define that the variable is readonly and the driver may allocate it in areas of memory faster to read. The function body simply calculates the index of the current thread and writes on the index of the output array the application of the lambda

function on the corresponding input element.

The Reduce kernel is more complex, but follows the same approach of having an inline OpenCL function as the lambda. The reduction happens on the kernel, and calls the lambda when merging two values.

The kernel itself is similar to the NVIDIA implementation described in subsection 2.4.4. The only difference is that the template is more generic than the specific NVIDIA implementation and supports more datatypes than the 4 supported by NVIDIA.

4.3 Map-Map and Map-Reduce fusion

Map and Reduce operations are, most of the times, used together. A common pattern is using map followed by a reduce. This is supported by ÆminiumGPU in an optimized fashion. Since both operations are independent for the programmer, a naïve implementation would execute them independently as well.

Figure 4.1 shows on the left the data transfers for that naïve implementation and, on the right, the data transfers for the same operations on the ÆminiumGPU.

Since data copies to and from the GPU are quite expensive, ÆminiumGPU implements a Map-Reduce fusion. The fusion of maps with other maps or reduces happens at runtime because there might be a branch condition in between that may cause a merging or not. Listing 4.2 shows one example of that, in which if `getInput()` returns 1, then only a map happens, else there is a map and a reduce that need to be merged.

Listing 4.2: Example of a Map-Reduce merge required at Runtime

```
1 PList<Integer> intermidate = input.map(ExampleMapLambda);
2 if ( getInput() == 1 ) {
3   Integer output = intermediate.get(0);
4 } else {
5   Integer output = intermediate.reduce(ExampleReduceLambda);
6 }
```

The solution for this is to use lazy evaluation. Instead of executing the operations inside the `map` method, they are delayed and stacked together until the results are in fact needed.

By default, Java is a strict language. That means that expressions are evaluated as soon as the Java Runtime sees them. In order to introduce laziness,

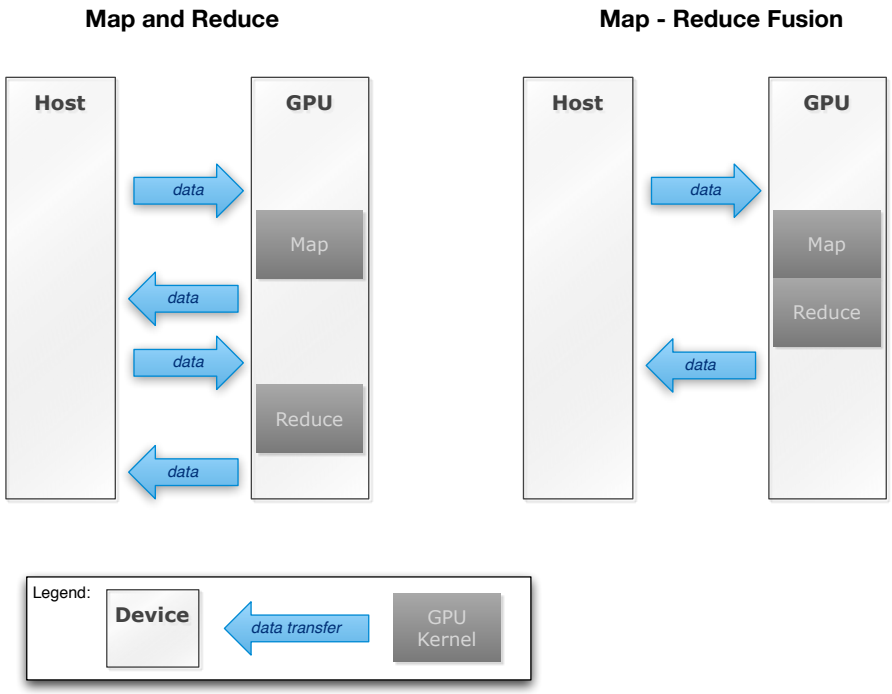

Figure 4.1: Representation of Map Reduce Fusion.

the `map` method returns a `LazyPList` that contains the information of that operation: a reference to the input array and the lambda to be executed.

Whenever a map is called on a `LazyPList`, the representation accumulates both maps, and returns a new `LazyPList` with a new lambda that is the composition of the two lambda functions. The pointer to the input data remains the same.

If a `reduce` method is called, then a special `MapReduceKernel` is compiled. That kernel is similar to the `Reduce` kernel, but whenever loading elements for the first time, it transforms them using the map lambda function. Reduces are not lazy, as they only return one value. There is no need to chain operations on just one value, as the parallelism of the GPU would not be used.

4.4 GPU vs CPU decider

As previously mentioned, several factors influence the speedup obtained by using the GPU. The decision to use the GPU or CPU for a certain operation is usually done by the programmer after benchmarking on each platform. However, for newly written code it is not possible to know beforehand which platform to use. ÆminiumGPU automatically decides which platform to use.

4.4.1 Related work

There has been previous work trying to balance execution between the CPU and the GPU. A common approach is to use information from previous executions.

One technique for real-time systems[19] is to collect the information from the last iteration of a loop to decided on the next. This allows for a real-time decision on the current operation.

Another work[14] extends this idea by taking into account the contention around the GPU when using data from previous executions.

Nevertheless, both these approaches require the code to have been previously executed, ideally with the same parameters.

A different approach[21] tries to use a cost-model to evaluate if a certain loop would be faster on the GPU or not. The platform retrieves data from micro-benchmarks, obtaining an average of the time each platform takes to execute a single instruction. Then, at runtime, it is possible to estimate the execution time for each loop, based on the following expressions:

$$\begin{aligned} Cost_{cpu} &= t_{cpu} \times insts \times output.size \\ Cost_{gpu} &= t_{gpu} \times insts \times output.size + copy \times \sum_{e \in inandoutput} e.size + init \end{aligned} \tag{4.1}$$

Equation 4.1 shows the cost model expressions. t_{cpu} and t_{gpu} are the average execution time for each instruction, *insts* is the number of instructions of the selected code. *output* is the output array generated by loop. *copy* is the time it takes to copy a certain size to the GPU one-way. *inandoutput* is the set of both input and output arrays. Since micro-benchmarks do not cover all possible sizes, a Least Squares regression is used to choose the parameters to use.

The difference between the GPU and CPU prediction models is that for the GPU, the time of data copying is also considered in both directions.

A limitation of this model is that it considers that all operations have the same cost on either platform. However, since the decision is based on bytecode, the difference in instruction times might not differ by much.

There is still one approach that tries to combine both platforms to perform the same operation[18]. The conclusion is that the slower platform will not help to improve the operation execution time. Therefore, only one platform should be used for a given operation. That does not mean that the other may not be doing other operations at the same time to improve the overall speedup of the application.

4.4.2 ÆminiumGPU Approach

ÆminiumGPU implements a decider based on a custom heuristic. Earlier tests have shown that the speedup of the GPU depends mainly on two factors: input size and operation complexity.

Other factors such as divergence and cache usage have also influence in the speedup, but are harder to detect either at compile time or runtime. Current methods to decrease the impact of such problems include profiling and posterior rewriting of code.

The complexity of operations can be determined at compile time, while the input size is only available at runtime. The reason for this is that input data may be read from disk or network IO or other operations may be performed before the target operation. In order to account for these factors, the decision is made at runtime using information from three different sources: micro-benchmarks, operation complexity and input size.

4.4.2.1 Micro-benchmarks

In order to predict which platform is best for each operation, one must have some information about the system. Whenever programmers start to use our platform, they run a micro-benchmark suite that records times for different operations.

Different operations are recorded, and for each operation different sizes are used. The sizes range from 10 to 10 million elements, and each element occupies 1 byte.

Each of the tested operations has a single instruction covering the majority of possible OpenCL instructions including arithmetic operations (+, −, ∗, /, mod), boolean operations (==, <=, >=, ...), mathematical functions (`sin`, `pow`, `log`, ...).

For each combination of input size and micro-operation, 30 runs are executed and different measures are recorded in a configuration file. For the CPU there is only the total execution time. For the GPU, the measures are more fine-grained and are divided into kernel compilation, memory transfer to the GPU, kernel execution and memory transfer back to host. This division is important because the memory transfer is the same and doesn't depend on the operation itself.

4.4.2.2 Operation Complexity

Since different operations take different times, it is important for the system to understand how complex the current operation is. ÆminiumGPU does this at compile-time. When generating the OpenCL version, it counts the usages of basic instructions. For instance, `sin(sin(x))` would have a complexity of $2 * sin$ in ÆminiumGPU. A larger operation would have more instructions and a larger complexity expression.

One of the main problems when predicting the cost model using static analysis is that it is not known how branching operations (`if`, `while`, `for`) will behave. In the case of `if`, both branches are considered, making a worst-case guess about the time. Since the metric will be applied to both CPU and GPU estimation, using both branches is conservative.

As for `while` and `for` loops, it is more difficult to guess how many iterations there will be. The approach is to use a representative value 20. The reasoning for this number is that 20 is enough to influence the decision towards parallelization. More accurate runtime-based models could have been used, but the cost of evaluating might be bigger than taking the wrong decision. Therefore the simple model was chosen.

4.4.2.3 Input Size

CPU Estimation

While previous parts of the process were done before executing the program, the estimation itself happens at runtime, where the input size is available. Before executing each operation, the prediction execution time for CPU and GPU is estimated and the one with faster time is chosen.

In order to estimate the CPU time, the micro-benchmark times are applied to the complexity expression. Since the platform only has times for round orders of magnitude, a exponential interpolation is used. The expression to calculate the final value between the values of the surrounding orders of magnitude is expressed in Equation 4.2. Auxiliary operations *bottom* and *top* are used to define the powers of 10 that limit the *value* argument. The regression is done by applying the logarithm to the exponential expression, and then obtaining the right values using linear regression to obtain the parameters of the

exponential expression.

$$
\begin{aligned}
bottom(value) &= 10^{floor(log_{10}(value))} \\
top(value) &= 10^{ceil(log_{10}(value))} \\
linear\_regression(x1, y1, x2, y2) &= (\frac{y2 - y1}{x2 - x1}, y1 - x1 \times \frac{y2 - y1}{x2 - x1}) \\
intpol(value, times) &= 10^{b} \times 10^{m \times value}, \\
where(m, b) &= linear\_regression\Big(\\
&\quad bottom(value), \\
&\quad \log_{10}(times(bottom(value))), \\
&\quad top(value), \\
&\quad \log_{10}(times(top(value)))\Big)
\end{aligned}
\tag{4.2}
$$

A complexity expression is a sum of parcels that have a number of repetitions for each operation. For the example code $sin(x) + sin(x)$, the complexity expression would be $2 * sin + 1 * plus$. Equation 4.3 shows the *CPUEst* estimation value based on the size and the n_{ops} pairs of repetitions (*reps*) and operations(*ops*). The estimation is the sum of the interpolation for each operation multiplied by the number of repetitions. $times_o$ is a function that, for each size power of 10, returns the time it takes to execute operation o.

$$
CPUEst(size, reps, ops, n_{ops}) = \sum_{i=0}^{n_{ops}} reps_i * intpol(size, times_{ops_i}) \tag{4.3}
$$

For instance, using the same expression $sin(x) + sin(x)$, the input size is 4500, the final estimation $CPUEst_1$ will be in Equation 4.4 where the $times_{op}$ function is loaded with the results of the micro-benchmark for that operation.

$$
CPUEst_1 = 1 * intpol(4500, times_{plus}) + 2 * intpol(4500, times_{sin}) \tag{4.4}
$$

GPU Estimation

The prediction for the GPU is more complex because using the GPU requires extra steps. These are listed below:

- Kernel compilation
- Memory copy from host to the GPU
- Kernel execution
- Memory copy from the GPU to Host

Kernel compilation: The variation in Kernel compilation time varies little with code complexity or size. Therefore, it is a constant (K_{compil}) retrieved from the previous micro-benchmarks.

Memory Copy from host to the GPU: The first memory copy is calculated using the same exponential interpolation mentioned above in a simplified way, represented by Equation 4.5. However, it might not be considered for the prediction if the input list belongs to the class `Range`. `Range` is a lazy initializer to the list, in which each element of the list will be created inside each thread on the GPU, based on the unique thread ID.

$$CopyHostDevice(size) = intpol(size, times_{copyHostDevice}) \tag{4.5}$$

Kernel execution: The Kernel execution depends on the complexity of the code. The more complex the code, the longer it will take to execute, and more instructions are executed. Furthermore, some operations take longer than others. The Kernel execution time is predicted using the same sum of interpolations for each unitary operation as used for estimating the CPU time. Equation 4.6 is similar to Equation 4.3 for CPU prediction.

$$KernelExecution(size, reps, ops, n_{ops}) = \sum_{i=0}^{n_{ops}} reps_i * intpol(size, times_{ops_i}) \tag{4.6}$$

Memory Copy from the GPU to host: The last memory copy is similar to the first, as seen in Equation 4.7. The sole difference is that the size argument differs whether it is a Map or Reduce operation. Maps have the same output size as the input size while Reduces always have the final size 1.

$$CopyDeviceHost(size) = intpol(size, times_{copyDeviceHost}) \tag{4.7}$$

Finally the GPU estimation is the sum of the estimations for each of these steps, describe in Equation 4.8.

$$GPUEst(size, reps, ops, n_{ops}) = K_{compil} + CopyHostDevice(size) + KernelExecution(size, reps, ops, n_{ops}) + CopyDeviceHost(size) \quad (4.8)$$

GPU versus CPU

Between GPU and CPU, the only difference lies in the extra steps required. In all other aspects, the estimation follows the same method. Since the resulting times are comparable, the platform with shortest time is selected.

Note that the estimation may not yield results approximate to the execution times. But, as long as both GPU and CPU model execution time on the same order of greatness, the estimations are acceptable. For example, if both estimations have an error of 20% above the real time, the decision is still correct.

5 Evaluation

This chapter is divided in 3 sections. The first section presents the usability tests performed. The second discusses the performance of the ÆminiumGPU. The last section of this chapter evaluates how accurate is the decision between GPU and CPU execution.

5.1 Usability Study

One of the main goals of the project is to provide an easy way to program for GPUs. A usability study was conducted in order to measure the difficulty of programming using the ÆminiumGPU platform. The main interest was to understand if the programming community in general was interested in using this approach to explore the potential in GPUs. Additionally, feedback about the programming experience is also useful to improve the platform.

5.1.1 Subject Profiles

The subjects for the study were 10 students of Informatics Engineering of University of Coimbra. Although a chocolate bar was announced as a reward for helping in the study, only one accepted it in the end. Others stated that they participated to help a scientific study and/or to learn new ways of programming.

All 10 students had a bachelor degree and from 3 to 8 years of programming experience. Figures 5.1, 5.2 and 5.3 represent the distribution of the students per expertise level of 3 different subjects: Java programming, functional style programming and OpenCL or CUDA programming.

Globally, subjects were at ease with programming in Java, and most of them had never tried to program for the GPU. Regarding FP, 4 students had

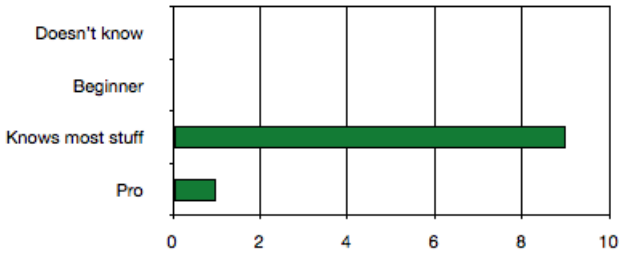

Figure 5.1: Distribution of expertise of subjects with Java

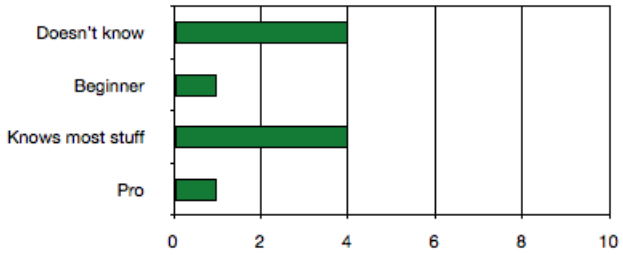

Figure 5.2: Distribution of expertise of subjects with Functional Programming

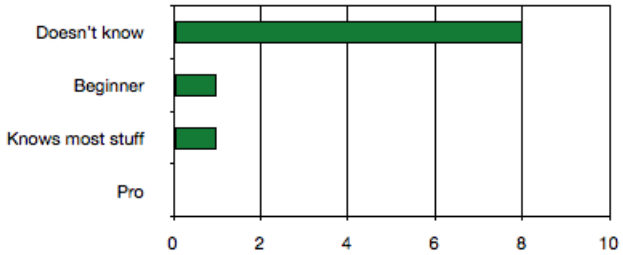

Figure 5.3: Distribution of expertise of subjects with Programming for GPU in Low level languages (OpenCL or CUDA)

never heard of it, and other 4 had done it in the past, although when questioned about it, they did not practice it regularly.

Regarding the environment, students were provided with a laptop or a desktop computer with a popular Java IDE. Students were provided with an initial code-base, with only some parts blank to be completed. This allowed for student to focus solely on the task at hand.

Before the execution of the Tasks, they were introduced to the ÆminiumGPU. The two-minute presentation focused on explaining map, reduce and how could these be combined together. The syntax was briefly described and

examples were provided.

5.1.2 Tasks

The Study was planned for 1 hour per subject, in order to exclude fatigue from thinking, programming and debugging for too long. Two tasks were proposed for that time. A MapReduce proficient programmer took 5 minutes to complete task A, and 10 minutes to complete task B. Since expected subjects did not have to master FP, the times were quadrupled to 20 and 40 minutes. The 4 times was a multiplier factor found reasonable to compensate the lack of knowledge of the framework.

5.1.2.1 Task A

> *"Find the sum of all natural numbers until 1000(inclusive) divisible by 7."*

Given the above assignment, subjects had to implement it first in regular Java and then using the ÆminiumGPU MapReduce library.

A MapReduce solution would be to start with a range of all integers until 1000. Then map all the numbers to themselves if they are divisible by 7. If not, map them to 0. Finally, reduce the result using the sum operation. An example of a solution is shown in Listing 5.1.

Listing 5.1: Solution for Task A using MapReduce

```
1 private static long gpuMapReduce() {
2   return new Range(10000).map(new LambdaMapper<Integer, Long>() {
3     @Override
4     public Long map(Integer input) {
5       return (input+1) % 7 == 0 ? (long)input+1 : 0L;
6     }
7   }).reduce(new LambdaReducer<Long>(){
8     @Override
9     public Long combine(Long input, Long other) {
10      return input + other;
11    }
12    @Override
13    public Long getSeed() {
14      return 0L;
15    }
16  });
17 }
```

5.1.2.2 Task B

> *"Find an approximation of the integral of the function $f(x) = e^{sin(x)}$ between 0 and 1 using the Trapezoidal rule[1]. Consider a resolution of 500 trapezoids."*

In task B, subjects also had to do one version in Java and another using MapReduce, but they were allowed to chose the order.

A possible solution using MapReduce would be to start with a range of 500 sequential integers. Then, for each one, calculate the area of the n^{th} trapezoid. In order to calculate that area, the boundaries of height are $n/500$ and $(n+1)/500$. The length of the parallel sides is given by applying the function f to each of the previous points.

To retrieve the final value, reduce all the areas using the `sum` function. An implementation of this solution is in Listing 5.2.

Listing 5.2: Solution for Task B using MapReduce

```
1 private static double gpuMapReduce() {
2   PList<Integer> li = new Range((int)500);
3   PList<Double> li2 = li.map(new LambdaMapper<Integer, Double>()
        {
4     @Override
5     public Double map(Integer input) {
6       double n = 500.0;
7       double b = Math.pow(Math.E, Math.sin(input / n));
8       double B = Math.pow(Math.E, Math.sin((input+1) / n));
9       return ((b+B) / 2 ) * (1/n);
10    }
11  });
12  return li2.reduce(new LambdaReducer<Double>(){
13    @Override
14    public Double combine(Double input, Double other) {
15      return input + other;
16    }
17    @Override
18    public Double getSeed() {
19      return 0.0;
20    }
21  });
22 }
```

[1]More information at `http://en.wikipedia.org/wiki/Trapezoidal_rule`

5.1.3 Analysis

The analysis of the study was divided in two components: one using a feedback form at the end of the session, and the other through observation of the performance of each individual.

The feedback form intended to retrieve feedback regarding how receptive programmers were to this approach, whether or not it felt as an easy approach for data intensive problems and how could it be improved.

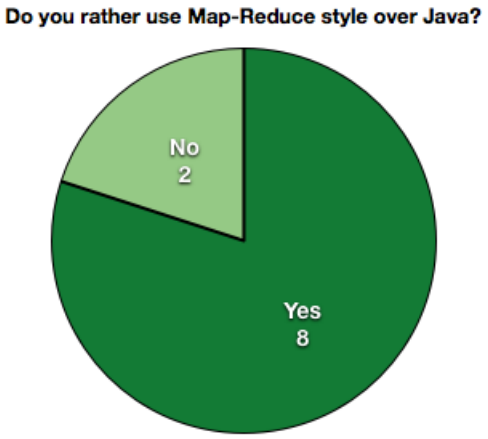

Figure 5.4: Distribution of opinions of participants whether MapReduce is more suitable than plain Java for these kind of problems.

Figure 5.4 reflects the preference of subjects regarding the two approaches taken. When questioned about their choices, reasons for choosing plain Java over MapReduce were that java felt more straightforward, without the need for knowing one extra API. Reasons for choosing MapReduce include the ability to run on both GPU and CPU and that, once learned, that style makes it easier to reuse programming patterns.

Figure 5.5 presents the answers of one of the main questions of the form. If using MapReduce would make the application run faster (a standard value of 2 times faster was given), would they use MapReduce. All the participants answered positively, which means that the overhead in learning this new style is not discouraging to use the platform.

Since GPU-programming is not yet mainstream, most subjects were not familiar with neither OpenCL nor CUDA, so answers in Figure 5.6 are not representative at all. Subjects that preferred MapReduce style liked the fact that they did not have to understand the underlying model or worry about how threads executed. The one subject that felt both had the same difficulty

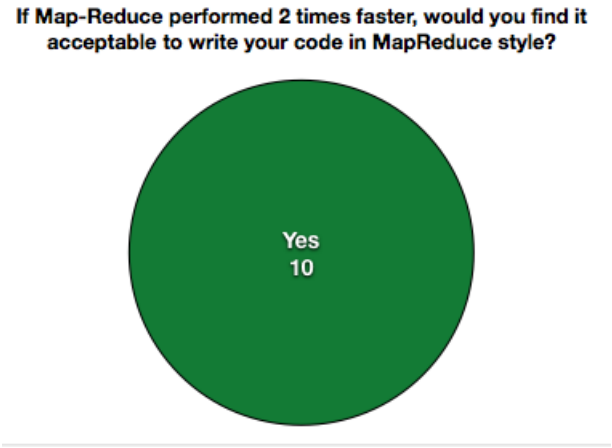

Figure 5.5: Distribution of participants that would choose MapReduce if it would perform two times faster.

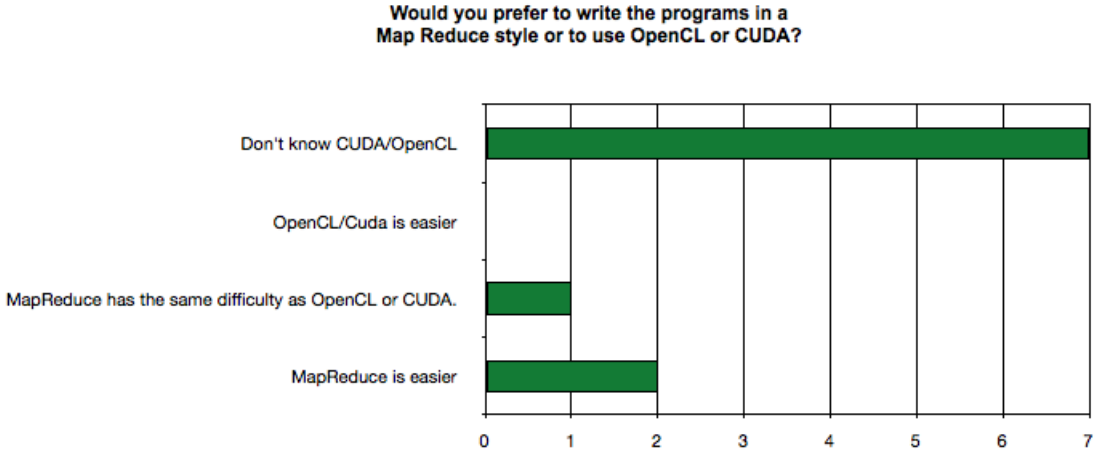

Figure 5.6: Distribution of the opinion of participants regarding Java MapReduce versus OpenCL.

justified that CUDA was not that hard once learned.

The last question, whose answers are in Figure 5.7, intended to understand how much of the Java syntax influenced the experience. Nearly all the subjects would prefer native support for lambdas, instead of using inner anonymous classes.

In fact, during observation, most subjects did spend some time figuring out how to use the right types inside lambdas. They were expecting the IDE (Eclipse in all cases) to provide auto-correction for types, which did not happen. This was revealed as a weak point in the experience.

Despite this issue, all subjects finished Task A successfully and 8 of 10

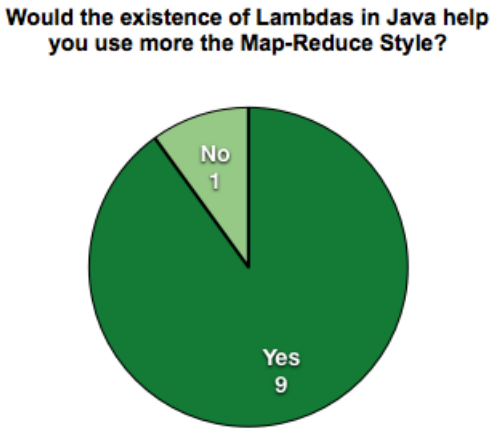

Figure 5.7: Distribution of the opinion of participants whether Java's inclusion of Lambdas would improve MapReduce writing style.

students did finish Task B correctly within the given time. Of the other two, one had taken a wrong approach and the other did not find one at all.

Task A was finished within 10 to 25 minutes by most participants. One subject did take 30 minutes for trying a wrong approach in the first place.

Task B was finished within 20 to 40 minutes by all subjects. The trapezoid area formula was provided upon request, so the major issue was implementing a MapReduce-based solution. All subjects but one thought of calculating each trapezoid area within the reduce phase, merging two values that limit the area. Within a few minutes, or after some experimentations, 8 of the subjects did realize that the second-level reduction would ruin the solution. At the second attempt, they would have had succeeded. The subject that did not do this mistake implemented the solution in just 20 minutes.

5.1.4 Summary

Due to the limitations of this study, it is not possible to generalize the conclusions to the global programmer community. It is however possible to understand the weak and strong points of the approach, based on this experiment.

Local junior programmers are interested in GPU programming that does not have a steep learning curve. They do find the MapReduce approach simple for data-intensive solutions. Although most of them were not familiar with the technique, they applied it by themselves within half an hour after a 10 minute explanation.

A weak point found is that the Java tooling is not ready to correct mistakes within anonymous inner classes, which should be improved within tooling sup-

port for Æminium/Java or by Java itself via a possible new lambda syntax in development (Java Specification Request 335, in progress at the time of writing).

This experiment also leads to the belief that programmers using ÆminiumGPU are able to write programs that use the GPU much faster than in traditional low-level languages such as CUDA or OpenCL. Subjects only modeled the problem itself in the map reduce framework. If they had to write the code in a low-level language, they would have to define how to write the program in parallel, when to copy data from host to device and vice versa, and finally decide how much parallelism to apply. These latter steps are avoided when using ÆminiumGPU.

Finally, it was concluded that programmers are interested in using the MapReduce style if it yields better performance.

5.2 ÆminiumGPU Performance

One important aspect of evaluating the performance of the ÆminiumGPU framework is to know how it behaves, when comparing to the CPU version and other alternatives.

5.2.1 Framework/Language Selection

In this experiment, a few examples were tested in different platforms. Using Æminium and plain Java are obvious choices, but it is also interesting to compare against other state of the art platforms. It was not possible to include Aparapi, although it would be interesting. Aparapi requires ATI hardware, which was unavailable. It was however possible to compare against a CUDA implementation, resorting to the MapReduce framework MARS [17]. Since CUDA is not directly comparable to Java, due to the JVM overhead, it was decided to include a C version as well.

5.2.2 Tasks

Three examples were chosen for evaluation. The first two examples are Task A and B of the Usability Study, mentioned before in Listings 5.1 and 5.2. The third example is calculating the minimum of function $f(x) = 10x^6 + x^5 + 2x^4 + 3x^3 + \frac{2}{5}x^2 + \pi x$ without using the derivative. It works by calculating the minimum for several points(the more points used, the better the precision) and selecting the minimum. An example of the implementation in ÆminiumGPU is in Listings 5.3.

Listing 5.3: ÆminiumGPU implementation of the fminimum problem

```
1 Double m = new Range((int)RESOLUTION).map(new LambdaMapper<
      Integer, Double>() {
2   @Override
3   public Double map(Integer input) {
4     double x = 2*input/(double)(RESOLUTION) - 1;
5     return 10 * Math.pow(x, 6) + Math.pow(x, 5) + 2 * Math.pow(x,
          4) + 3 * x * x * x * x + 2/5*x*x +Math.PI * x;
6   }
7 }).reduce(new LambdaReducer<Double>(){
8   @Override
9   public Double combine(Double input, Double other) {
10    return Math.min(input,other);
11  }
12  @Override
13  public Double getSeed() {
14    return Double.MAX_VALUE;
15  }
16 });
```

CPU versions, both in C and Java, were written using the Accumulator pattern, which saves memory space in sequential programs.

Each of these tasks was executed with a different size of input data. A logarithmic scale was used because the interest lies in how the performance varies with the order of magnitude. For each order of magnitude, 9 values were tested, all equally spaced. For instance, for order of magnitude 10^2, tests were run with 100, 200, 300, 400, 500, 600, 700, 800 and 900 of size. For other orders of magnitude, the same approach was applied. The size is the number of elements of a float array used as input.

5.2.3 Setting

Experiments were all run on the same machine. The CPU was a dual core Intel Core2 Duo E8200 at 2.66GHz, backed by 4GB of RAM memory. The GPU is an NVIDIA GeForce GTX 285, with 240 CUDA cores. The machine runs Ubuntu Linux 64bits with the NVIDIA CUDA SDK 4.0 with OpenCL 1.0.

All tests were executed 30 times. In order not to include the effects of caching, the order of execution was alternated and the garbage collector was forced between executions.

Despite running each test 30 times, it is not possible to take conclusions

with a statistical significance, since there is no guarantee that all machines behave like this, or even that the times follow a normal distribution. Nonetheless, the experiments work as indicators of how well the platform works on a random GPU/CPU pair. Depending on what machine the tests are run, results will surely be different.

5.2.4 Results

The final execution times, in seconds, can be seen in Figure 5.8. The graphs on the left have the C version in blue versus the CUDA MARS in green. The graphs on the right have the Java version in blue and the ÆminiumGPU version in green.

The top two graphs are for the SumDivisible example, the middle two for Integral and the bottom two are for the Function Minimum (*FMinimum*). Note that the x axis is using a logarithmic scale.

Despite expecting the MARS version to be faster than the C version, it is actually always slightly slower, and the difference increases with high order of magnitudes.

ÆminiumGPU starts as being slower than the Java version, but around 5000 elements, it starts to be faster. Also, it does not have such as exponential increase as the Java version.

All platforms scale up to 10 million elements except for MARS for the integral example. The reason for not having the corresponding values is that it exceeded the memory limit on the graphics card.

There are two main reasons for MARS not having a speedup similar to ÆminiumGPU. First, MARS uses a key-value MapReduce similar to Hadoop. Although it is beneficial to several algorithms, there is an overhead of at least 2 times in memory copying, which is a big part of operation, taking up to as much time as the reduce phase all together. ÆminiumGPU optimizes for simple datatypes and copies only the necessary.

The second reason, and less important, is that MARS calls three kernels on the GPU. One for mapping, other for grouping values with the same key, and another one for reducing. ÆminiumGPU optimizes MapReduce sequences and merges into only one kernel, which executes faster.

5.2.5 Summary

For two lightweight examples of mathematical utility, it is possible to see that ÆminiumGPU has a good performance for large arrays. For the Integral exam-

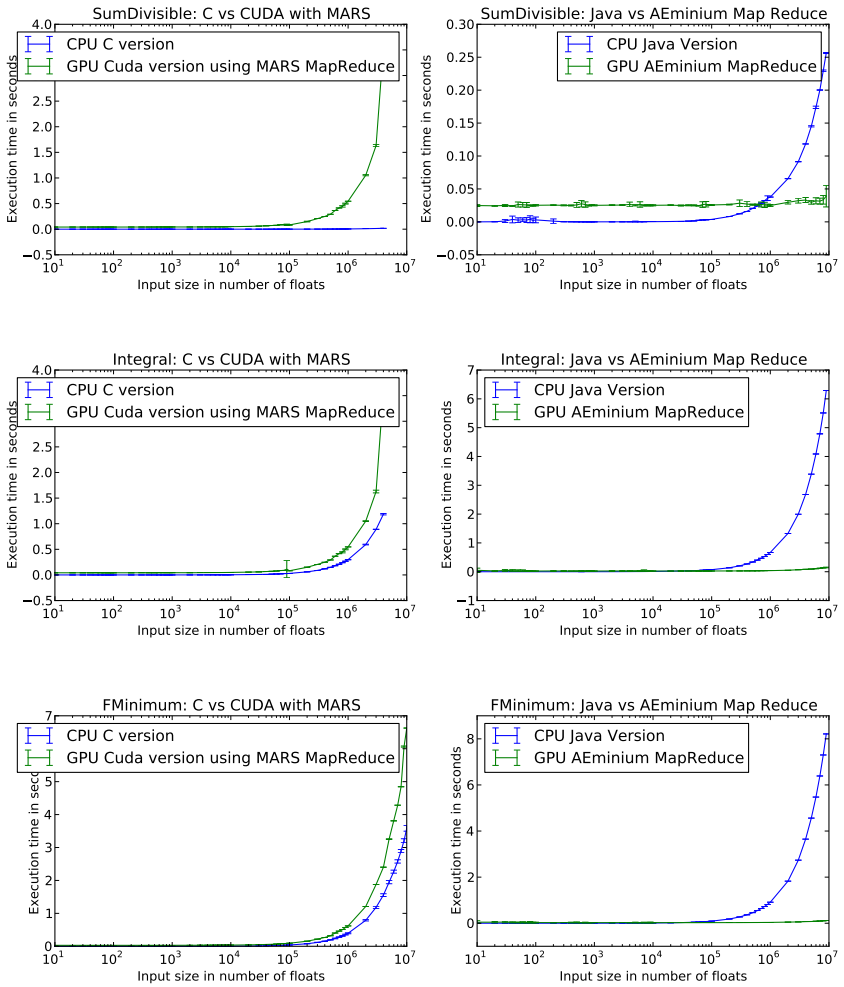

Figure 5.8: SumDivisible, Integral and Function Minimum examples in C and CUDA and in Java and Æminium.

ple it achieves a speedup of 42 times for 10M elements over the iterative Java version and of 70 times for the Function Minimum under the same conditions.

These values are a good indicator of performance of the library on operations that follow the map-reduce pattern.

5.3 GPU-CPU Decider

One of the aspects where ÆminiumGPU needs to be evaluated is on how it handles the decision of CPU vs GPU. Since programs can be executed on both platforms, the ÆminiumGPU Runtime tries to guess which one will be faster. Some examples were benchmarked to evaluate how accurate the decision is.

The setting of this experiment was the same as the one used when evaluating the ÆminiumGPU performance versus other alternatives.

5.3.1 Tasks

For this experience, five operations were chosen to be executed. The first two are simple operations, $map(sin, xs)$ and $map(\lambda x : sin(x) + cos(x), xs)$ where xs is a given list with a certain size.

The third operation is more computation intensive: $map(fact, xs)$. For each element of a sequential list of integers, it maps each element to the factorial of itself. For large numbers this computation is intensive despite being made of mostly multiplications. The implementation of $fact$ was iterative and not recursive, since OpenCL does not support recursive functions and it would have worse performance.

The last two operations are the same used in the ÆminiumGPU performance evaluation: Integral and Function Minimum.

5.3.2 Results

Figures 5.9 to 5.13 plot the results from the experiments. Again, the input size varies across orders of magnitude, just like in the previous experiment, hence the logarithmic scale is also used in the x-axis. Since in many cases, time increases in a linear way, the logarithmic scale was used in the y-axis.

Each figure contains two related graphs that share the x-axis. The top graph shows four colored lines: two for prediction and two for execution times. GPU and CPU expected time (`cpu_exp` and `gpu_exp`) are the values predicted by the platform before executing the program. GPU and CPU real time (`cpu_real` and `gpu_real`) are the actual times it took to execute the

operation. Each of the values on the graph is an average of the values obtained from 30 independent measurements.

On the same graph, there is a black line which follows a different scale (on the left side of the graph). This line represents the accuracy of the framework for a given size. If the platform guesses correctly 15 out of the 30 runs, it will have an accuracy of 0.5. This is necessary because running times are not constant and can vary due to external factors such as the CPU scheduling, the behavior of the GPU controller, latency on the memory copies, and so on.

On the bottom graph, there is another measure of quality. The red line represents the time lost by using the GPU when using the CPU would be faster. The blue line represents the time lost by taking any wrong decision, either GPU when CPU would be faster or vice-versa.

This measure is important to complement the accuracy. The time lost by taking the wrong decision may — or may not — be important for the user to evaluate this solution.

5.3.3 Analysis

The first two examples (Figures 5.9 and 5.10) are similar, changing only in a matter of scale. The GPU is always faster and the system predicts accordingly. The number of runs where the framework failed to correctly guess was very reduced. And, the error was never greater than a few mili-seconds.

The few runs that fail to guess correctly are few and miss by a few mili-seconds. Those cases in which the GPU appeared to be faster, but it was slightly slower.

For the Factorial example (Figure 5.11) the same observations are true, but the curves start to increase earlier. It is also possible to notice that the predicted curves evidence an exponential behavior in steps between orders of magnitude. The cause for this is that the exponential regression is obtained just from the two adjacent powers of 10. If the whole range was used, the curve would fit better the real time, but it would be slower to compute the regression.

The Integral example (Figure 5.12) shows a different behavior. The accuracy is nearly none until 5000, but from then on it increases and stabilizes at 1. The CPU expected times are much higher than the real times. The model fails in this case. Reasons for that might include some low-level CPU optimizations (caching, SMID instructions, etc).

The time lost by wrongfully using the GPU can go up to the 45ms, which accounts for several times the CPU execution time. This is common only

for orders of magnitude below 10^4. For such input sizes, the decider is too expensive to be even used. Calculating the regressions and fetching micro-benchmark values from disk are operations too expensive compared with the small examples tested. As such, it was concluded that the decider would not be used for input sizes below 10^4 elements.

The Function Minimum example (Figure 5.13) is similar to the Integral, having the same accuracy behavior, with lost times on the GPU varying on the same scale.

5.3.4 Summary

As reflected on the graphs, there is a great disparity on the values of the execution time for different input sizes. Programs can take around 50ms for 10 elements to 290.000ms for 10 million elements.

Although the overall accuracy is 77%, above the mark of 7000 elements, accuracy is always 100%. On the other hand, the time lost in a wrong platform for smaller inputs can represent several times the of execution on the fastest device.

However, the wrong decisions below 10^4 are not visible to the user because the system has a conservative approach for small arrays. The calculations involved for the decision are not worth executing below that limit, hence the CPU is always used.

Overall, the performance of the decider is good, not only in terms of accuracy, but specially in respect to the overall lost time.

Finally it is important to mention that the values presented here are specific for the machine in which tests were run. In a different machine, the behavior can be different.

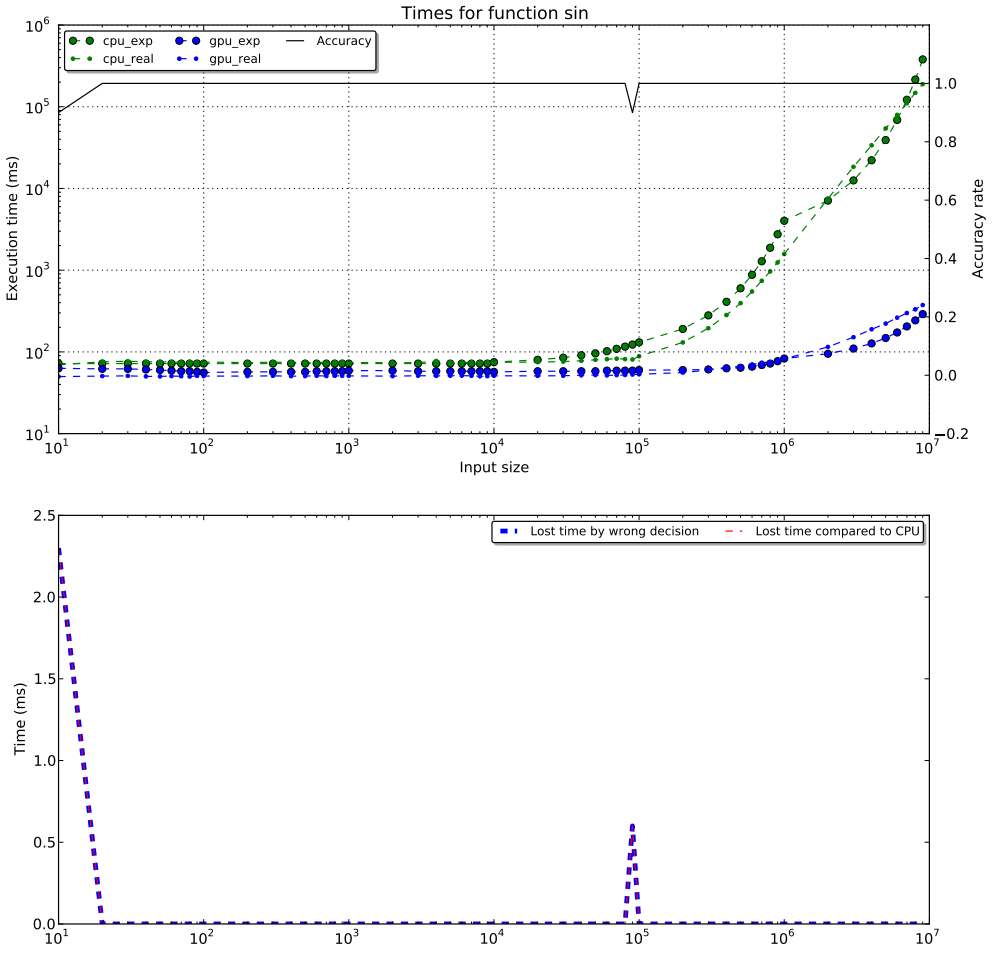

Figure 5.9: Prediction results for $map(sin, list)$.

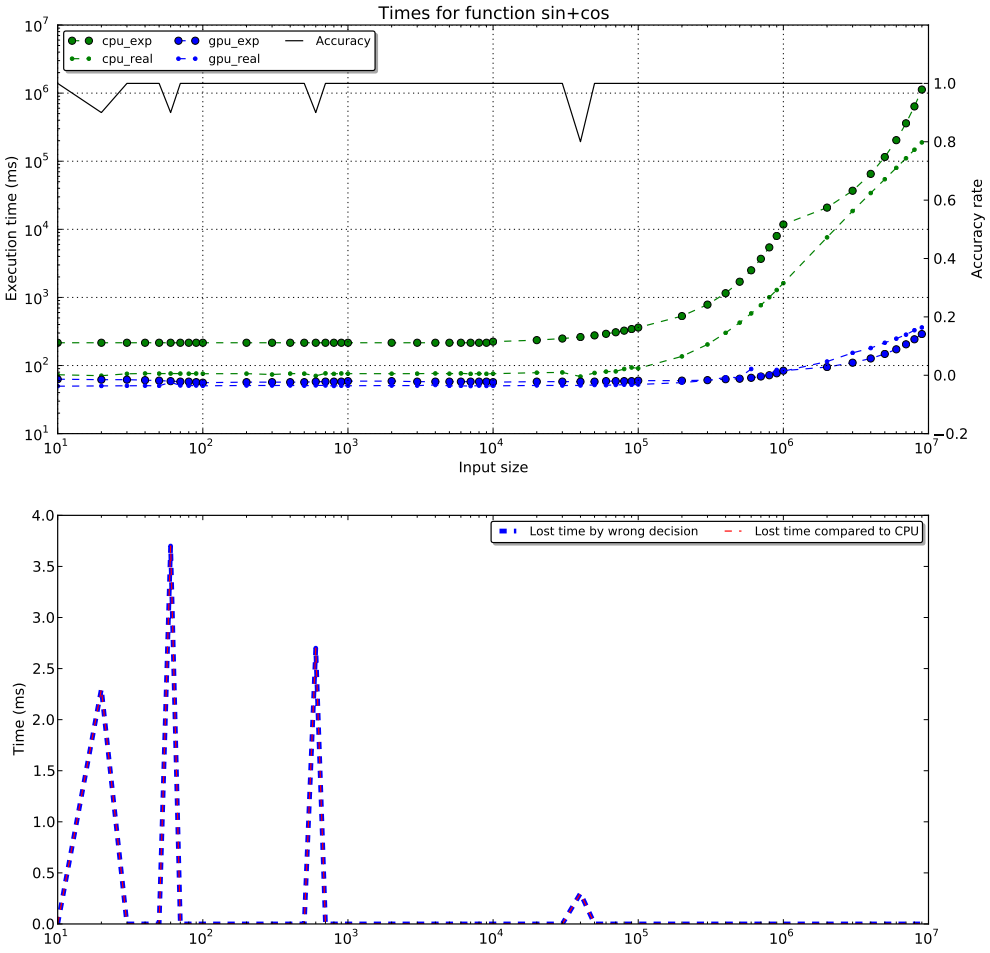

Figure 5.10: Prediction results for $map(\lambda x : sin(x) + cos(x), list)$.

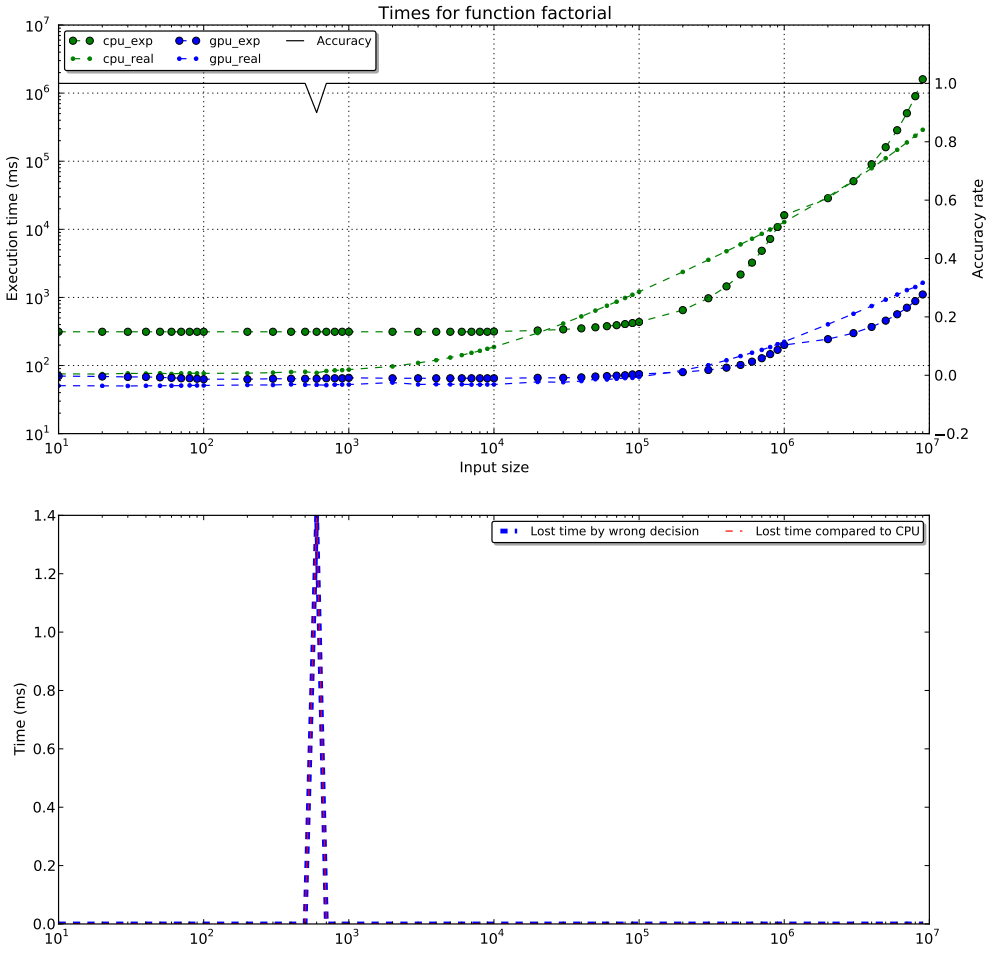

Figure 5.11: Prediction results for $map(fact, list)$.

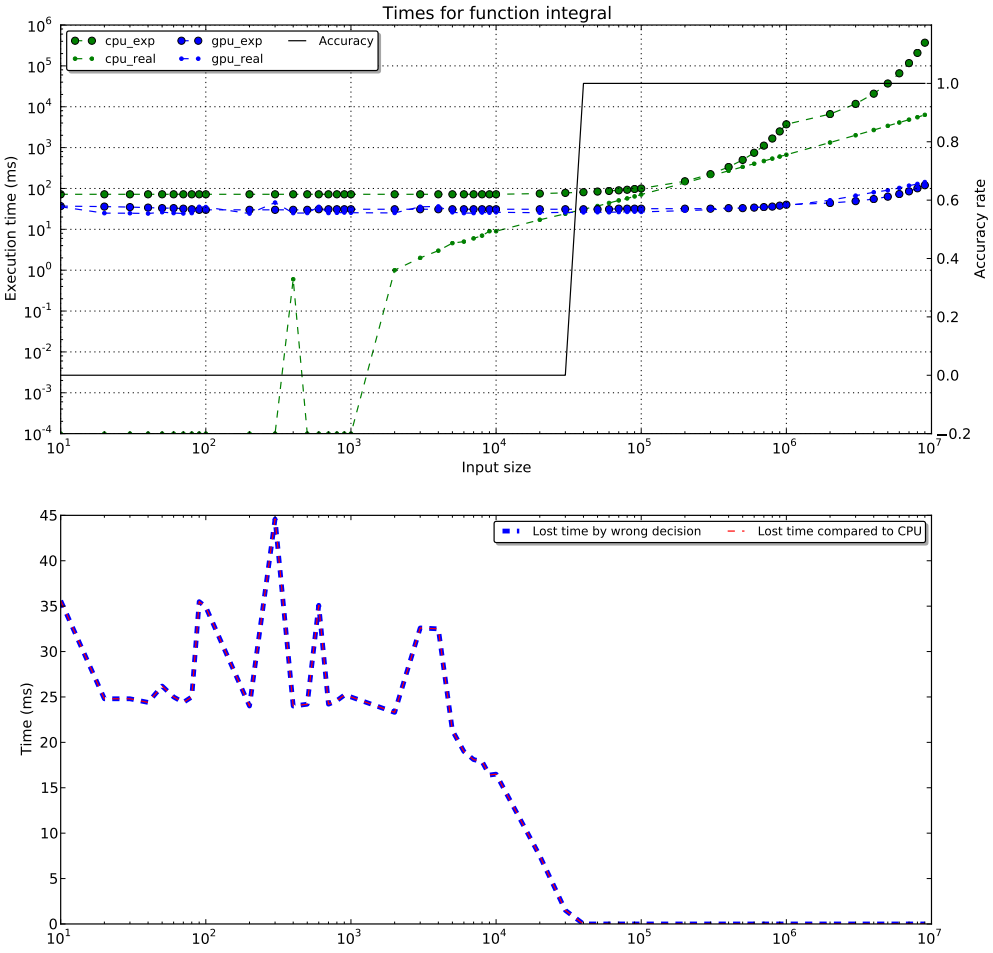

Figure 5.12: Prediction results for Integral.

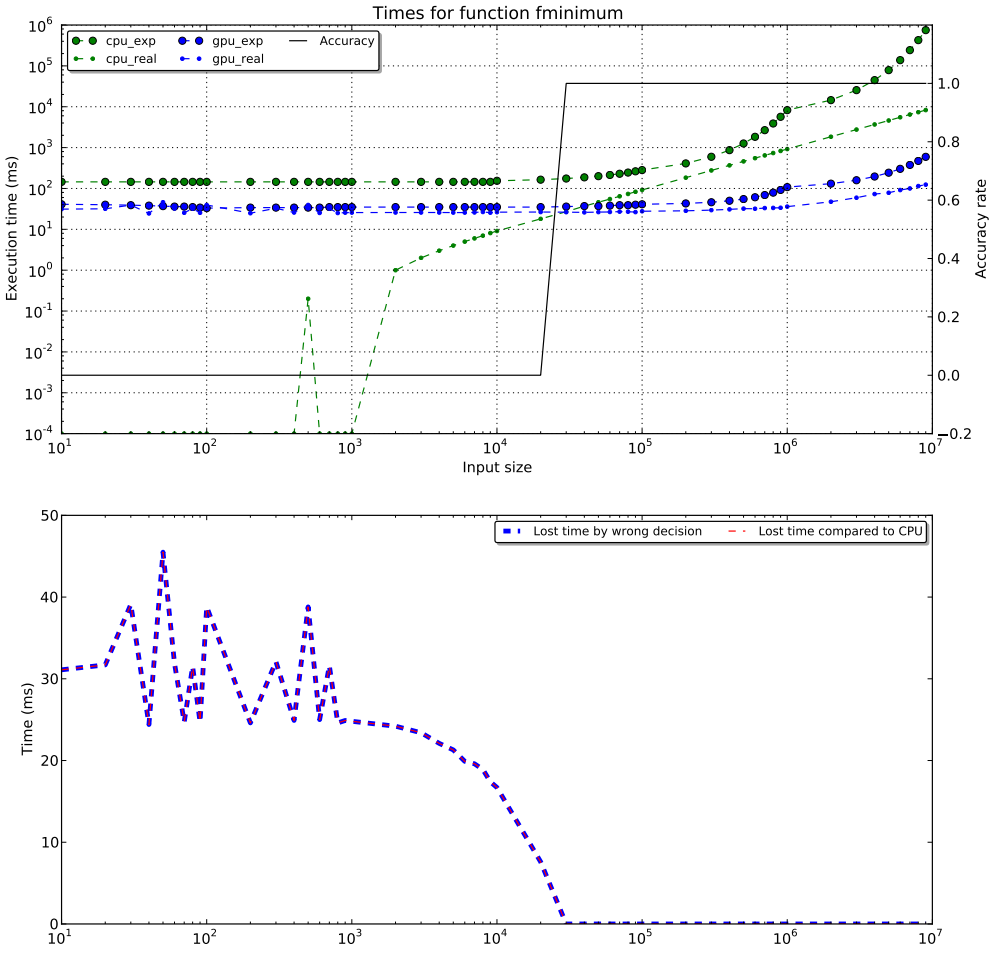

Figure 5.13: Prediction results for Function Minimum.

6 Future Work

This chapter is organized in three sections. The first discusses how this work can be expanded with more features. The second section focus on the integration with the Æminium system. Finally, the last one concerns future research paths.

6.1 Extension of ÆminiumGPU

As mentioned in the previous chapters, ÆminiumGPU only supports map and reduce operations. However, it is extensible for other operations and such work is expected to happen in the future.

For one-dimensional lists, it would be interesting to implement operations such as *filter*, *find*, *zip with* and others.

Filter takes a list and a predicate, a function that returns True or False for a given element, and returns a new list with all elements that validated the predicate.

Find is an operation that takes a list and a value and returns index of the first (or alternatively all) of the elements that are equal to the given value. For example: $find(3, [0, 3, 4, 1, 3])$ would return 1 since the second element is the first 3 in the list.

Zip with is an operation that takes two lists and returns a new list with pairs of elements from each list. For instance, the operation $zipWith([1, 2, 3, 4, 5], [5, 0, 0, 0, 1])$ would return $[(1, 5), (2, 0), (3, 0), (4, 0), (5, 1)]$. This operation would need a new kind of lists for vectorized types, such as pairs of two numbers.

In order to work with matrices — a basic unit in numerical calculations — it is planned the implementation of a Matrix data structure with specific operations. These operations include the existing map and full reduce, but

also partial reductions for lines and columns of the matrix.

In addiction, matrix operations such as sum and multiplication are also planned as they appear commonly in larger problems.

6.2 Integration with the Æminium Language

Given the lack of a compiler for Æminium, the approach of applying the Compiler to the Java Language meant that ÆminiumGPU does not support the Æminium language itself right now. Future plans are to adapt the ÆminiumGPU Compiler from Java to Æminium, making use of the already existing compiler.

This adaptation would be a one-to-one translation from Java AST to Æminium AST with extra functionalities for syntactic sugar which Æminium might have by that time.

A final step on the integration would be to adapt objects and methods to be part of the Æminium standard library, in order to make these parallel collections used either by default or very easily for any Æminium programmer.

6.3 Future Research

In order to make the integration as smooth as possible for the programmer, it is important that the selection of GPU or CPU is done automatically and as accurately as possible.

Despite current achievements in accuracy and lost time, there is room for improvements on the decision algorithm. Incorporating other factors such as thread divergence and caching would benefit the current approach.

Another open question is how well the prediction will work on other platforms. Due to unavailability of hardware, not all GPU architectures or families were tested. For instance, ATI GPUs might behave differently.

Since GPGPU is in a fast paced expansion, new hardware is being frequently released. For instance, AMD and ATI have already shown their next generation architecture, Fusion. Fusion has a GPU and a CPU on the same die, named a APU for Accelerated Processing Unit. This new design avoids bottlenecks related to the north bridge in communication between CPU and GPU. This new design may pose new challenges by changing the rules on what makes a program faster on either platform.

6.4 Summary

The available operations in ÆminiumGPU are only a subset of what can be done with this system. Adding more operations and datatypes is something that is planned.

This project has two moving targets: the Æminium language and the GPGPU platforms. It is a challenge to keep the ÆminiumGPU project updated with both targets. Additionally, it is important to keep the balance between GPU and CPU programming experiences.

7 Conclusions

This chapter draws the conclusion of the document by summarizing the work done, reviewing its importance and discussing how it has met its expectations.

7.1 Overview

ÆminiumGPU is a platform for executing data-parallel programs on heterogeneous platforms that include both a CPU and a GPU.

The system provides programmers with a compiler and a runtime system. The compiler detects parts of code that can be executed on the GPU and generates the low-level code for the Kernel. The runtime is responsible for executing the code and applying additional optimizations.

The API exposed to the programmers is based on collections of elements, in this case, lists that have two parallel operations: map and reduce.

Map and reduce were chosen because of their parallel nature and how well they can be used to implement algorithms together.

Given the natural coupling of these operations, the platform optimizes consecutive calls and merges them, saving transfer time from and to the GPU. The platform also includes other optimizations such as lazy generators which allow the generation of sequential lists on the GPU, saving expensive memory copies.

Another optimization in the compiler is the compilation of binaries for common operations, even though they might not get to be executed. Although that costs time in the compile-phase, it saves time during execution.

The usage of the GPU on mathematically-based examples can go up to 70 times, which is proof that the system does improve performance.

Finally, for a given operation, the runtime attempts to guess which platform will be faster. This decision has 100% accuracy with elements above 7000 elements for the tested programs, which prevents the system for wasting much time on the CPU, when it could be using the GPU.

The overall accuracy is only of 77% due to the lack of precision for smaller elements. However, since the decider costs more computation power than the execution of the program for those smaller sizes, the lost times because of the GPU will not happen.

7.2 Relevance

Discrete programmable GPUs are now part of commodity hardware and have the potential to improve the performance of application. However, to make use of that potential, the available approaches are too low-level and require a deep understanding of GPUs inner works.

ÆminiumGPU lowers the barrier by allowing programmers to write data-parallel algorithms in a high-level approach (MapReduce) and in a high-level language (Java and in a near future Æminium). And it does so by minimizing the overhead of the abstraction.

For general purpose Java programmers, this means that, within few minutes of learning the API and MapReduce style, they can be programming for GPUs. ÆminiumGPU not only reduces the learning curve, but it also makes the actual programming for GPUs faster by doing all the necessary routine tasks under the hood.

The implemented building blocks, Map and Reduce, are used in several types of applications. They are widely used in machine learning, financial analysis, biomedical applications, signal processing, computational biology and chemistry, among many others.

ÆminiumGPU allows for domain-specific programmers to use the GPU without knowledge of the different programming model. This was confirmed in a field test with 10 junior programmers. Subjects managed to complete a simple example that used most of the capabilities of the system within half-and hour. The feedback provided confirmation of the usability of the solution.

7.3 Final remarks

Considering the requirements elicited at the beginning of this thesis, it is possible to conclude that the goals set for this dissertation were all successfully accomplished.

A framework was designed and implemented in order to allow developers to use GPUs straightforwardly, not much differently from how they use the CPU. Furthermore, ÆminiumGPU was made available to the public[1] and extensively evaluated, always evidencing good results in performance and usability.

[1] Available at `https://github.com/alcides/AeminiumGPUCompiler/`.

Bibliography

[1] Javacl library. http://code.google.com/p/javacl/. http://code.google.com/p/javacl/ [accessed 13-January-2011].

[2] Scalacl. http://code.google.com/p/scalacl/. http://code.google.com/p/scalacl/ [accessed 16-June-2011].

[3] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[4] Gpu market share report. http://jonpeddie.com/publications/market_watch/, 2010. [accessed 29-November-2010].

[5] Opencl programming guide. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_OpenCL_ProgrammingGuide.pdf, 2010. [accessed 29-November-2010].

[6] Apple. Opencl parallel reduction examples. http://developer.apple.com/library/mac/#samplecode/OpenCL_Parallel_Reduction_Example/Listings/reduce_float_kernel_cl.html. [accessed 13-January-2011].

[7] ATI. Ati radeon hd 5970 specifications. http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5970/Pages/ati-radeon-hd-5970-specifications.aspx, January 2010. [accessed 5-September-2010].

[8] D. Behr. Amd gpu architecture. Presented at PPAM 2009, September 2009.

[9] A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley. Hadoop: a framework for running applications on large clusters built of commodity hardware. *Wiki at http://lucene.apache.org/hadoop*, 2005.

[10] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23:777–786, August 2004.

[11] B. Catanzaro. Opencl optimization case study: Simple reductions. http://developer.amd.com/documentation/articles/pages/

opencl-optimization-case-study-simple-reductions.aspx, 2010. [accessed 17-June-2011].

[12] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. Mcdonell, and V. Grover. Accelerating Haskell Array Codes with Multicore GPUs. 2011.

[13] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[14] C. Gregg, J. Brantley, and K. Hazelwood. Contention-aware scheduling of parallel code for heterogeneous systems. In *2nd USENIX Workshop on Hot Topics in Parallelism*, HotPar, Berkeley, CA, June 2010.

[15] M. Harris. Optimizing parallel reduction in cuda. http://developer.download.nvidia.com/compute/cuda/1_1/Website/Data-Parallel_Algorithms.html/, 2010. [accessed 17-june-2011].

[16] M. Harris, S. Sengupta, and J. Owens. Parallel prefix sum (scan) with cuda. *GPU Gems*, 3(39):851–876, 2007.

[17] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[18] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. Mapcg: writing parallel program portable between cpu and gpu. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 217–226, New York, NY, USA, 2010. ACM.

[19] M. Joselli, M. Zamith, E. Clua, A. Montenegro, A. Conci, R. Leal-Toledo, L. Valente, B. Feijó, M. d. Ornellas, and C. Pozzer. Automatic dynamic task distribution between cpu and gpu for real-time systems. In *Proceedings of the 2008 11th IEEE International Conference on Computational Science and Engineering*, pages 48–55, Washington, DC, USA, 2008. IEEE Computer Society.

[20] I. Kozin. Working notes of distributed computing group - intel 'nehalem' processor. http://www.cse.scitech.ac.uk/disco/publications/WorkingNotes.Nehalem.pdf, April 2009.

[21] A. Leung, O. Lhoták, and G. Lashari. Automatic parallelization for graphics processing units. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 91–100, New York, NY, USA, 2009. ACM.

[22] D. Luebke, M. Harris, J. Krüger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, and A. Lefohn. Gpgpu: general purpose computation on graphics hardware. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, New York, NY, USA, 2004. ACM.

[23] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22:896–907, July 2003.

[24] M. D. McCool, Z. Qin, and T. S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[25] NVidia. Cg faq page. `http://developer.nvidia.com/object/cg_faq.html`, 2010. [accessed 29-November-2010].

[26] NVidia. Geforce3 series product page. `http://www.nvidia.com/page/geforce3.html`, 2010. [accessed 29-November-2010].

[27] NVidia. Tesla c1060 computing processor board specification. `http://www.nvidia.com/docs/IO/43395/BD-04111-001_v05.pdf`, 2010. [accessed 5-September-2010].

[28] C. Omar. Atomic hedgehog: Productive high-performance computing with python. Presented at NVIDIA Research Summit 2010, 2010.

[29] R. Pawlak, C. Noguera, and N. . Petitprez. Spoon: Program Analysis and Transformation in Java. Rapport de recherche RR-5901, INRIA, 2006.

[30] A. Prokopec, T. Rompf, P. Bagwell, and M. Odersky. On A Generic Parallel Collection Framework. Technical report, 2011.

[31] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. Citeseer, 2007.

[32] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr Dobbs Journal*, 30(3):http://www.gotw.ca/publications/concurrency–ddj.ht, 2005.

[33] P. M. Sven Stork and J. Aldrich. Concurrency by default: using permissions to express dataflow in stateful programs. In S. Arora and G. T. Leavens, editors, *OOPSLA Companion*, pages 933–940. ACM, 2009.

[34] H. Wiki. Differences between hlsl and glsl. `http://hl2glsl.codeplex.com/wikipage?title=Differences%20Between%20HLSL%20and%20GLSL`. [accessed 23-January-2011].

[35] R. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 198–207. IEEE, 2009.