Extending Scala with General Purpose GPU Programming

A Dat 6 report by:

Reidar Beck
Helge Willum Larsen
Tommy Jensen

Supervised by:

Bent Thomsen

JULY 29, 2011

AALBORG UNIVERSITET

Student report – *not* an official document

Aalborg University
Department of Computer Science
Selma Lagerlofs Vej 300
9220 Aalborg
Telephone:(45)96358080
http://www.cs.aau.dk

Title:
Extending Scala with General Purpose GPU Programming

Theme:
GPGPU, parallelization

Project timeframe:
Dat6, February 1. 2011 - July 29. 2011

Project group:
FLD606A

Group members:

Helge Willum Larsen

Reidar Beck

Tommy Jensen

Supervisor:

Bent Thomsen

Abstract:

In this report we document an attempt to make it easier to use powerful GPUs by extending the Scala compiler to automatically ofload work to the GPU.
We benchmark similar code and find that it provides between 2-3 speedup compared to the CPU alone.
Finally we discuss ways to improve the extention to ofload more work.

Copies: 5

Total pages: 84

Appendices: 0

Paper finished: 29th of July 2011

Preface

This report was written during the spring semester 2011 by group FLD606A.

The intended audience of the report are people who have at least the same general level of knowledge in computer science as the authors, as well as some experience with compiler theory, although little knowledge of GPU programming is assumed.

A note on numbering In this report figures, tables and equations have a number of the form $x.y$ where x is the page on which they were inserted and y is a unique number for that page. This should make it easier for the reader to locate the reference in question, as it will be found on either that or the next page.

We would like to thank:

Bent Thomsen for supervising throughout this project.

Contents

1
Introduction

In this report we build on the work we did during the previous semester[BLWJ] trying to make it easier for developers to write programs that use the powerful Graphics Processing Units which many modern computers are now equipped with, so that we can increase the speed of general, non-graphic related, computations.

This is necessary as GPUs are very different from CPUs, due to CPUs being mostly designed to do diverse computations on data in serial and GPUs are designed to do the same computation on data in parallel. This means that in order to take advantage of the new hardware capabilities, we have to either retrain programmers or develop a way to automate the translation from the way programmers are used to write for the CPU to the way the programs should run on the GPU.

In the last report[BLWJ] we documented a number of tools to make it easier to write code for the GPU. Here we document the development of a plugin for the Scala compiler which is capable of translating certain code patterns to run on the GPU.

We choose to develop the plugin for the Scala compiler as it already has got a powerful plugin system and because much Scala code is written in a functional style which minimises side-effects, and that makes it easier to run it concurrently.

1.1 Motivation

In 2005 Herb Sutter published the article “The Free Lunch Is Over” [Sut05] which stated that the CPU progress we have observed over the past many years may come to a halt. With single-threaded code, one could expect it to run faster with every new generation of processors, due to increased clock speed and execution optimisation. As illustrated in figure 10.1, since the year of 2005 there has not been any significant increase in clock speeds. This is due to thermal limitations in present semiconductor chip designs.

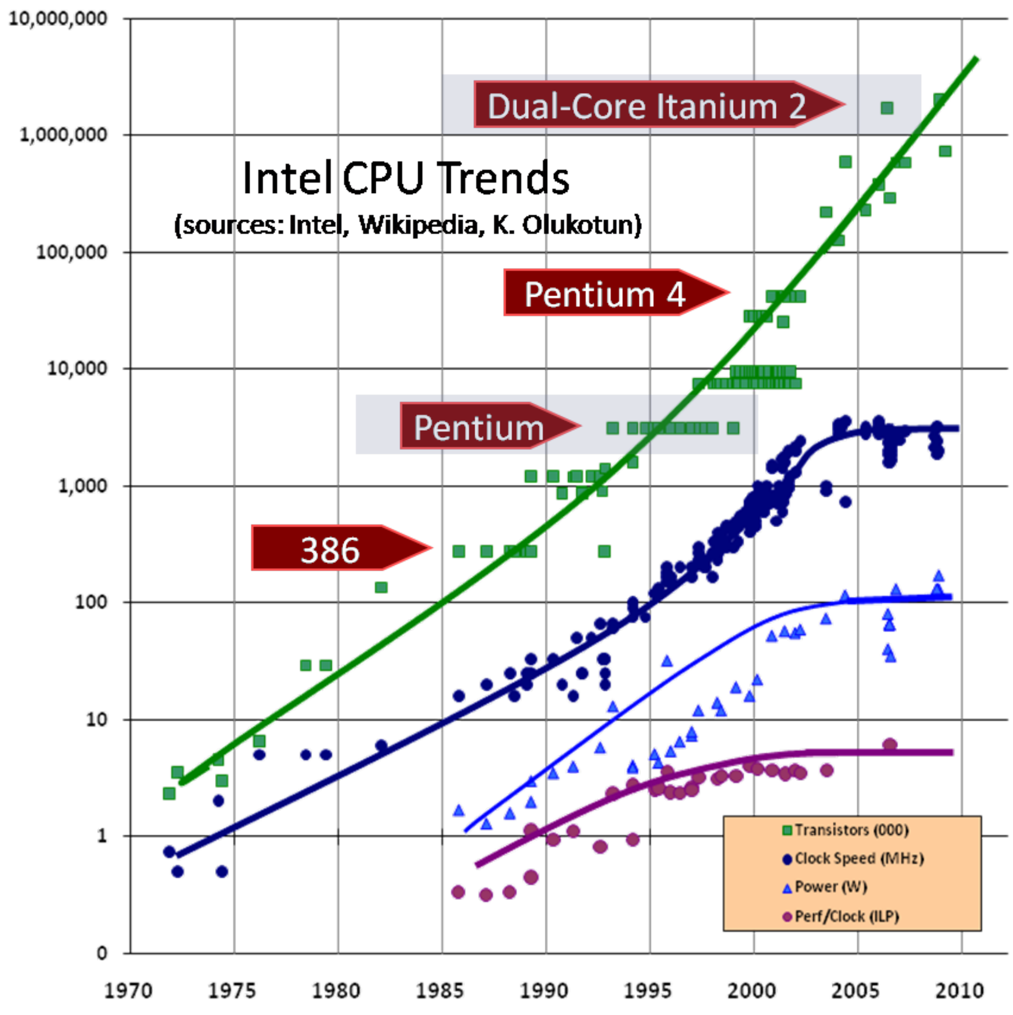


Figure 10.1: Intel CPU trends [Int07]

Moore's law states that transistor density is growing at an exponential rate [Moo65] and will double roughly every two years. This law is still in effect and is expeced to be so for many years to come. This has led CPU vendors to divert focus from higher clock speed to fitting more CPU cores on to a single die, leading to many-core CPUs.

In 2011, scientists at University of Massachusetts Lowell made an experimental CPU of one thousand cores on a single chip [fas11]. It will take a number of years before we see so many cores in mainstream computers, however, it gives us an insight as to what the future CPUs will look like.

Recently, with the introduction of OpenCL [ope], programmers have been allowed to write general purpose code that runs on mainstream graphics cards. With most mainstream desktop/laptop computers, having GPUs with thousands of simple SIMD (Single Instruction Multiple Data) cores that can utilise massive throughput with parallel applications, they can also provide an insight on how the future of writing applications will potentially be.

To write applications to the GPU, OpenCL provides a low-level C API where memory management is handled by the programmer. Considering the difficulty of making concurrent applications and the low-level OpenCL language, it will be challenging for the "traditional" programmer to take advantage of this extra processing power.

In order to ease the transition of writing concurrent applications we will search for a solution to have "traditional" programmers take advantage of the GPU in a familiar setting without having to learn a new programming language or radically changing their ways of making applications.

Part I

Analysis

This is a world of action, and not for moping and droning in.

Charles Dickens

2 GPU

2.1 GPGPU Platform

Graphics Processing Units were originally introduced for the single purpose of accelerating 2D and later 3D graphics computations on their targeted platforms. Since the beginning of the 2000's, with the introduction of programmable shaders in OpenGL and DirectX, GPUs have evolved from special purpose hardware into a more general purpose programmable architecture. Today it is commonly seen that physics simulation, scientific computing, video encoding/decoding and other highly parallel tasks are accelerated on this recently emerged General Purpose GPU platform. The two leading vendors of GPU hardware, NVIDIA and AMD, even have products specifically aimed at general purpose computation, with no display port for connecting an external monitor.

When consulting documentation and professional articles about the GPGPU platform, different concepts and terms are used for describing and presenting the underlying architecture. This depends on the goal of the very documentation, hardware vendor and/or API and the perspective of the targeted reader. For software developers, there are reference documentation for the different programming APIs; these stick to the specifications of the API in question, but the concepts and layout of the programming architecture may vary from and even collide with terms of the underlying hardware architecture or other programming APIs. This ambiguity introduces confusion when attempting to explain what the GPGPU platform looks like, as the description of it depends on the point of view.

2.2 GPU Architecture

At first glance, the GPU architectures of the two leading manufacturers, AMD and Nvidia, look very similar. They both brand their products for how many shaders/cores, global memory, clock speed and FLOPS (FLoating point OPerations per Second) they have. The lower-level details as to how their branch prediction works are still trade secrets and thus not explained in their specifications. Architecture changes with every generation, however, the focus upon these two vendors seems that whilst AMD uses many simple ALUs (Arithmetic Logic Unit) with relative low clock frequency, Nvidia tries to use fewer yet more complex ALUs[Bit11].

As an intermediate programmer making an attempt at GPGPU programming, these lower-level details might not be of mere relevance, but are nevertheless important to be aware of as they can severely impact performance depending on which application is written. As an example: if you are running an algorithm based on SHA-256, it makes heavy use of the integer rotate-right operation. On an AMD card, this operation is executed in a single hardware instruction whilst on an Nvidia card it requires three separate hardware instructions to emulate (two shifts + one add)[Bit11].

Even though GPUs from these two vendors vary in their architecture, they have a common standard for listing features and specification we can use for optimising OpenCL code. This standard notion is called Compute Capability and is, at the time of writing, at version 2.0.

As Nvidia explain their features in CUDA terminology and AMD use OpenCL terminology, table 16.1 lists the matching terms for which the names differ between the two terminologies.

Table 16.1: OpenCL and CUDA terminology

Cuda term	**OpenCL term**
GPU	Device
Multiprocessor	Compute Unit
Scalar core	Processing element
Global memory	Global memory
Shared(per-block) memory	Local memory
Local memory	Private memory
kernel	program
block	work-group
thread	work-item

2.3 OpenCL

OpenCL is a framework which enables you to write programs that can be executed on heterogeneous platforms possibly consisting of CPUs, GPUs and other processing units.

It was initially developed by Apple, and in collaboration with big companies including NVIDIA, AMD, INTEL and IBM, an initial proposal was forwarded to the Khronos Group in the hopes of defining a new standard for making a cross platform environment for general purpose programming on the GPU. Khronos was finished with the 1.0 specification on December 8th 2008 [ope] with aid from representatives from different companies. As of now, the latest version is 1.1 and was finalised on June 14th 2010.

The OpenCL framework provides a low-level C like programming language for writing kernels, and an API for defining and controlling platforms. A kernel is a piece of code to be executed on the platform, and the platform is a collection of OpenCL enabled devices such as host CPU, GPUs and possibly other types of hardware.

OpenCL is defined in three types of models, which are: platform, execution and memory.

2.3.1 Platform Model

As different hardware have different platform models, OpenCL provides an abstract platform model view, presenting a unified view to heterogeneous hardware. Thus, the programmer merely has to write one version of his code and the runtime takes care of translating it to the different hardware platforms.

In figure 18.1 the OpenCL platform model is shown.

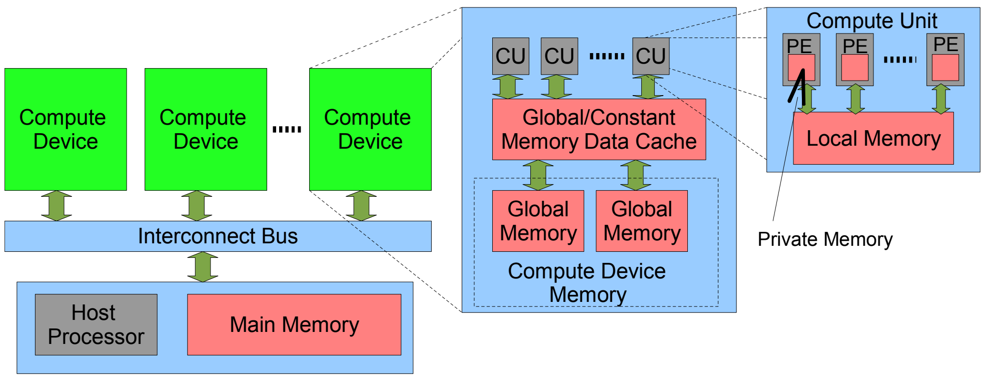


Figure 18.1: OpenCL Platform Model

The host processor is mapped to the host CPU whilst the main memory is the RAM of the particular computer running the OpenCL host code. They are connected by a bus to one or more Compute Devices being OpenCL enabled e.g. CPUs, GPUs, stream processors. These Compute Devices are further divided into CUs (Compute Unit), having a memory hierarchy which is elaborated on later. The Compute Units on a CPU are mapped to the different cores. On a GPU the Compute Units are mapped to the number of multiprocessors.

The CUs are once again made up of PEs (Processing Element)—virtual scalar processors operating on one thread at a time. On a GPU each PE is mapped to a single core in a multiprocessor, whereas on a CPU it can be comparable to an ALU (Arithmetic logic unit) inside one of the cores where private memory is mapped to the ALU registers.[Coo10]

2.3.2 Execution Model

Execution of OpenCL code occurs in two parts: kernels that run on one or more OpenCL compliant devices and a host program executed on the host. The host program defines a context where the kernels execute. This context includes the following resources:

- Devices: Collection of OpenCL enabled devices
- Kernels: The OpenCL functions that run on devices
- Program Objects: The program source and executable that implement the kernels
- Memory Objects: A set of memory objects visible to the host and OpenCL devices. Memory objects contain values that can be operated on by kernels

When a context is created, it can be manipulated by the host by using functions from the OpenCL API. To coordinate the execution of the kernels on the devices, the host creates a data structure called command-queue. The host then places commands on these command-queues which are then scheduled onto the devices within the context. A command can be of the following kinds:

- Kernel execution commands: Execute a kernel on PEs of a device
- Memory commands: Transfer data to and from devices; map memory objects
- Synchronisation commands: Synchronise the execution of commands

When a kernel is sent by the host to a device for execution, an index space is made covering a number of work-groups. Each work-group executes on a single CU and work-groups are further composed of many work-items which are executed in parallel on the PEs of a CU.

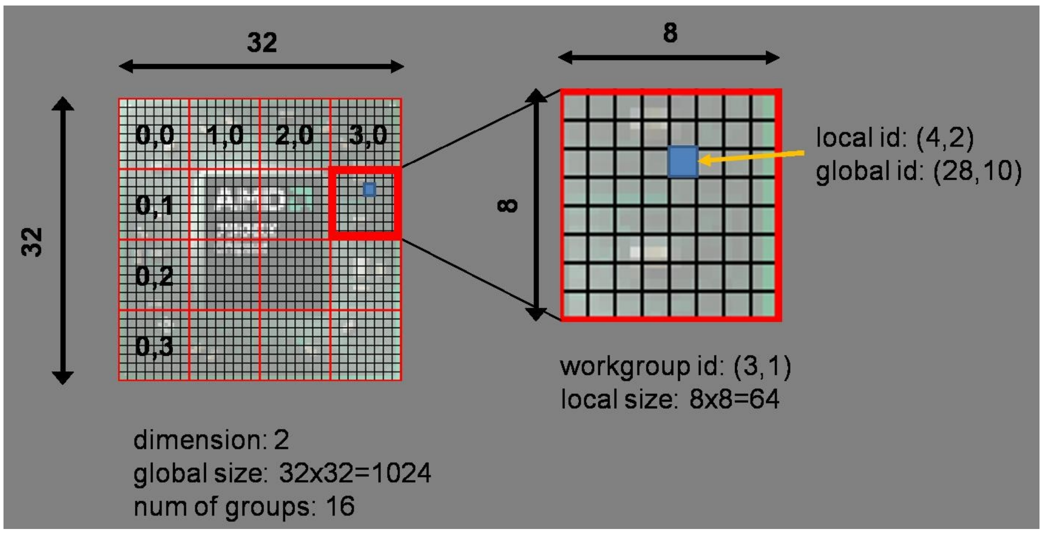


Figure 19.1: two-dimensional OpenCL work-group and work-items [amd]

Figure 19.1 shows a two-dimensional kernel with 16 indexed work-groups. Each of these work-groups includes 64 indexed work-items. The highlighted work-item has a local id of (4,2). It can also be addressed by its global id by using the highlighted work-group offset of (3.1) by multiplying it with the work-group dimension length and adding the local id.

2.3.3 Memory Model

The memory model view is, as illustrated in picture 20.1, a hierarchy of memory banks that are:

- Global Memory: Allows read/write operations to all work-items in all work-groups.
- Constant Memory: Allocated by the host and is constant during the whole execution of a kernel.
- Local Memory: Local to a work-group and shared by work-items.
- Private Memory: Region private to a work-item.

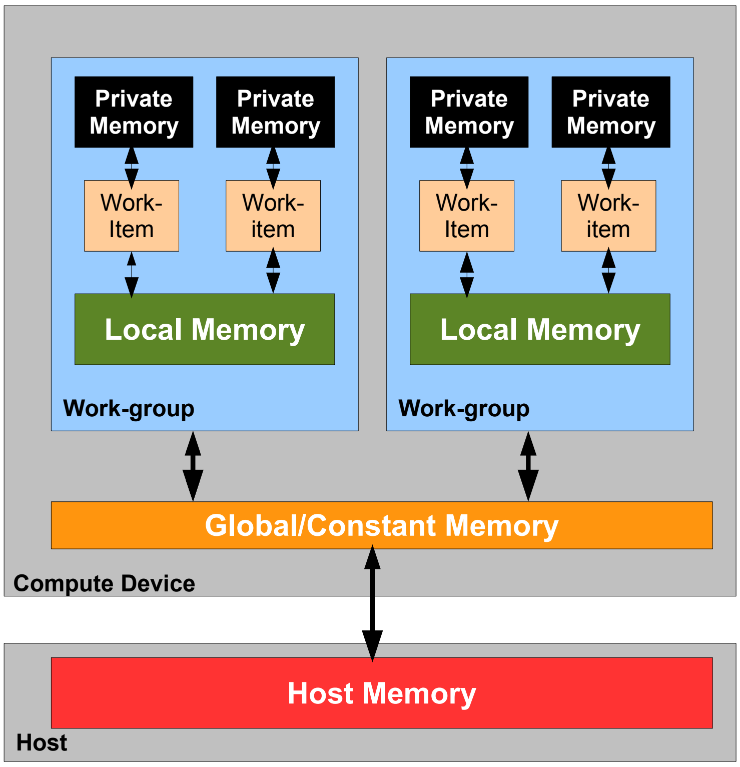


Figure 20.1: OpenCL Memory

When writing kernels in OpenCL the memory must be declared with certain address space qualifiers to indicate whether the data resides in global, constant, local or private memory, or whether it will default to private within a kernel.

3 Alternatives

3.1 Domain Specific Language

As we have previously seen, a GPU is normally programmed in C or C++. This is an issue, as neither C nor C++ are specifically designed to handle the issues that may arise in a massively concurrent environment. Because of this, both languages lack the capability to describe the operations which the GPU kernels typically are used for.

For example, C has an addition operator, so one can write:

```
int cc;
addAndAssign(int a, int b){
  cc = a + b;
}
```

And cc would have the expected result. However, when we deal with the GPU we often write code which deals with arrays instead of single numbers, yet if we try to write:

```
int[10] cc;
addAndAssign(int[] a, int[] b){
  cc = a +b;
}
```

we get an error.

At the end of the day, we want a language where we can write that and have the result of the element-wise addition to be stored in the cc array. This would be possible if we had a language that was tightly coupled to the specific domain of GPU programming. Such a language would not contain the features from C that cannot be used on the GPU, such as function pointers or memory allocations, but would have arithmetic, e.g. matrix multiplication, build in.

While we could write an entire new language from scratch this would be a vast amount of work and also presents us with a potential problem as that language would have been designed to run on the GPU and not the CPU. Instead of doing this, we could extend an already-existing language. Since C++ allows us to override operators, we could define a class GPUArray with the proper code to do the calculations on the GPU. Then we could write:

```
GPUArray cc;
addAndAssign(GPUArray a, GPUArray b){
  cc  = a + b;
}
```

and have it executed as we want.

The downside of this approach is that it is limited by how flexible the underlying language is. C++ is sufficiently flexible to allow overriding operators but not to define new operators, and we cannot easily convert existing arrays to GPUArrays as arrays in C++ are pointers to a memory area and we are unaware of know the size of the allocated memory.

The advantage is that it is relatively simple to write the DSL, and programmers who already master C++ can be expected to comprehend it faster than an entirely new language.

3.2 Functional Programming

Where Object Oriented Programming is focused on objects, functional programming is focused on functions. In functional programming languages, functions are first class values; that is, they can be given as arguments to other functions, returned from functions and stored in variables. In addition, functional programs are often written in such a way that they do not modify variables, but instead preferring to return a new value that is the result of applying the function to the input. Functions which only depend on their input and always return the same result given the same input, without changing any global variables, are said to be pure[Has11].

Much of the inspiration from functional programming has been derived from the lambda calculus, a mathematical model of computation devised by Alonzo Church[Chu41]. Under the lambda calculus, computation is modelled as evaluating a mathematical expression. This is also one of the reasons many of the functions used in functional programming are pure, since no system of mathematical equations will ever assign a variable two different values.

In addition, much of functional programming is spent working with (linked) lists – they are available in the standard library of Java and several other Object Oriented languages, but they are rarely used as they do not offer cheap random access (like arrays do), nor do they have a particularly good cache performance. However, linked lists are better suited to functional programming because they save space (and work): a pure function, which takes a list as input and returns the sum of the elements of the list prepended to the list, does not need to copy each element (as would be necessary with an array, because there might not be space in front of the elements)—it needs only to return the first element with the reference to the next element set to the head of the original list.

While there are many different things one can calculate, functional programs often use two patterns: creating a new list which contains the result of applying some function to each element of an input list, and calculating the result of applying some function to its previously returned result and each element in the list.

The first pattern is almost always available as the map function, which takes its name from the idea that mathematical functions (such as the square root) are maps from one set of elements to another. Thus, to calculate square roots of the first ten natural numbers, one could write:

```
val sqrtList =  List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).
```

```
   ↪      map(math.sqrt)
```

The second pattern seems more complicated, but, in fact, it is not. Consider that we now have the list of square roots and we want the sum of them. This is a situation where we want to apply the addition function to each element in the list, plus the previous sum.

If we assume that the addition function is `add`, we can write this as:

```
val sum = sqrtList.reduce(add)
```

In this case, there are ten elements in the list, so we could also have expanded the calculation to become:

```
val sum = add(add(add(add(add(math.sqrt(1), math.sqrt
   ↪      (2)), math.sqrt(3)),  math.sqrt(4)), math.
   ↪      sqrt(5)), ... )
```

3.2.1 Functional Programming and Concurrency

As elaborated on in the section of GPU architecture2.2, GPUs are massively parallel. This is in contrast to CPUs which typically have four cores or less and, until recently, often only had one.

This is a problem because most of the popular languages in use at present, according to the TIOBE index[TIO11][1], are not designed with concurrency as a basic principle. This means that it is up to the programmer to write the program in such a way that it can take advantage of multiple CPU cores.

However, multi-threaded programs are prone to a number of bugs such as race conditions, dead-lock and live-lock. These bugs can be very difficult to detect and resolve as they only manifest when some specific non-deterministic condition occurs.

As an example, the following Java code may or may not be safe to

[1]Java, C, C++ and C#

run concurrently, depending on whether any other thread modifies the counter variable while it is being accessed in the loop:

```
for(String item : ArrayOfItems){
  counter += veryExpensiveFunction(item);
}
```

yet this is not very obvious from the code.

Pure functions can always safely be run on multiple cores as long as their input data does not overlap. If Java had first class functions we could rewrite the previous example in a functional style (pmap is assumed to be a function that loops through the array in parallel on all the computers cores):

```
ArrayOfItems.pmap(veryExpensiveFunction).sumList
```

Naturally, `map` is simple to parallelise as long as the given function is pure, because the function does not depend on the value of any other element in the list. On the contrary, `reduce` is more complicated to parallelise. Since later calls to `reduce` depend on the previously calculated values, we can only parallelise the calls if the function given as the argument is associative[1]. This is the case for some functions such as integer addition, multiplication and string addition, but not others, such as subtraction or division. It is trivial to show that proving a function to be associative is undecidable, so we will have to use a less general way to decide whether a given instance of `reduce` can be parallelised.

Even if the function is associative, we will still have to synchronise the calculations each time we have reduced the current level.

3.3 Scala

Scala is a functional and objective oriented programming language designed by *Martin Ordersky*. The Scala compiler targets the Java virtual machine and Scala can be used with almost any existing Java library.

[1]That is, the order in which the function is applied does not matter.

In regards to our project it is, however, much more important that the Scala compiler has a plugin system that allows us to add additional compiler phases, and that we can alter the Scala code before it is compiled down to JVM instructions.

Compiler optimisations are no news and most compilers optimise a number of things, such as constant expressions, register usage, method in-lining and stack allocation. What is so great is that these optimisations can be used without the programmer having to be considerate of them or having to rewrite the code.

The downside to this is that the compiler plugin can only get the information available in the source code, and that it must not change the semantics of the code. Since it is undecidable whether two pieces of code have the same effect, the compiler plugin will be unable to optimise every case and will almost certainly produce code that is slower than a human could have optimised it to.

3.4 ScalaCL

ScalaCL is an approach to have people who master Scala to make use of OpenCL without having to learn C or a new language. Initially, ScalaCL was an internal Scala DSL with limited parallel expressions on GPUs. It created OpenCL kernels out of its internal AST representation and executed them through OpenCL4Java OO (Object Oriented) bindings, OpenCL4Java being a library used for calling OpenCL's C API.

```
import scalacl._
import scalacl.ScalaCL._

class VectAdd(i: Dim) extends Program(i) {
    val a = FloatsVar // array of floats
    val b = FloatsVar
    var output = FloatsVar
    content = output := a + b
}

var n = 1000;
var prog = new MyProg(n)
prog.a.write(1 to n)
prog.b.write(prog.a)
prog !
println prog.output
```

Listing 29.0: ScalaCL DSL vector multiplication

Listing 28.1 shows how a simple multiplication of two arrays looks like in ScalaCL DSL.

The ScalaCL DSL is no longer maintained as the author thought it had little or no resemblance to original Scala, as the whole idea of making ScalaCL was to make people who master Scala able to use OpenCL without having to learn a new language. With the ScalaCL DSL language, people knowing Scala had to learn a new DSL language [Oli11] instead. In addition, with a DSL there is little or no control structure or support of structured data.

Realising the limits to ScalaCLv1, all effort has now been shifted to ScalaCLv2. ScalaCLv2 includes two things:

- Scala compiler plugin.

  The compiler plugin is used to optimise general loops on arrays, lists, and inline integer ranges. These loop optimisations do not rely on any library or hardware, as loops in Scala are an order of magnitude slower than equivalent while-loops. For-loop performance has been addressed many times on Scala forum and development page but no major change has been made since 2008 [Sca11c].

- Library of OpenCL-backed collections

  The ScalaCL collections make use of the compiler to translate Scala functions into OpenCL code.

```
import scalacl._
import scala.math._

implicit val context = new ScalaCLContext
```

```
val r = (0 to 1000000).cl
val a = r.toCLArray
val m = a.map(v => cos(v).toFloat)
println(m)
```

Listing 30.0: ScalaCL version 2

As seen in listing 29.1, ScalaCLv2 resembles native Scala far more than ScalaCLv1 does and should therefore be easier to use for Scala programmers. Having ScalaCL collections can easily be used directly with Scala iterators like map, flatMap, filter and foreach, as shown in Listing 29.1.

ScalaCL looks promising but is still Alphaware and has more bugs than features acording to the developer [Oli11]. These are the collections that are currently supported:

- CLArray[T]
  - Supports types of *AnyVal* and tuples og *AnyVal*
  - asynchronous OpenCL implementation for map, filter and clone operations
  - OpenCL implementation for min, max, product, sum operations
- CLFilteredArray[T]
  - Same types as CLArray[T]
  - asynchronous map and filter
- CLFunction[A, B]
  - A,B can be of types tuple or AnyVal
  - a CLFunction can be created explictily but it is reccomended to write it as a Scala function and let the compiler plugin do the translation

3.5 Extending Jikes

There are two places where we can implement compiler optimisations for languages targeting the JVM – in the compiler that compiles it down into Java byte-code or in the JIT[1] compiler which turns the Java Bytecode into executable code for the current hardware platform.

Each approach has its own benefits – if we optimise the language compiler we will work with the code at a higher conceptual level and can then take advantage of the semantics offered by it, but optimisations to the JIT compiler means that any language targeting the JVM can use the optimisations. This matters for the end user as well, because if the JIT compiler is optimised, the user will then have to install the optimised version in order to get the benefits, whereas if we optimise the compiler the user gets the speed benefits, but this will only work with programs that have been compiled with the enhanced compiler – old programs will not get any faster.

Either approach is valid, and the JIT optimisation has been achieved before with Jikes.

Jikes is an open source compiler for the Java language, made by IBM[IBM11a]. Due to Java source code not being compiled to native code but to a virtual instruction set, it is necessary to have a virtual machine execute the resulting code. To satisfy this need, IBM has also worked on the Jikes RVM (*R*esearch *V*irtual *M*achine)[IBM11b].

Unlike the Java virtual machine made by SUN, the Jikes RVM is not meant to be used to run Java, but as a vehicle to enable researchers to do research in how to create and improve virtual machines; the result is a virtual machine easy to extend but also not having the full Java class library. This might be a problem if we wanted to develop a complex application, but for our research purpose it being easier to extend is of greater importance.

Finally, Jikes RVM has a rather unique feature for a Java Virtual Machine: it is primarily written in Java rather than a more common choice for virtual machines such as C or C++. This allows people who are more

[1]Just In Time

used to working with Java to improve a Java virtual machine, without having to learn a new programming language.

Extending Jikes RVM was precisely the choice made in [LLL09]. The central idea in the paper is to automatically translate loops (primarily loops through arrays, but in principle any loop that can be run in parallel without changing the result) to code which gives the same result.

The paper placed the loops in the Java class files into three different categories[LLL09]:

GPU implicit loops These are loops from which the work can be spread out so that every turn through the loop is executed in parallel across the processing elements of the GPU. This makes the loop implicit. If there are enough processing elements to handle each iteration the cost of the loop is reduced to the cost of transferring the data, plus the cost of one loop iteration.

GPU explicit loops These are loops which are also run on the GPU, but where the entire loop runs on the same processing element. This is only performed if the loop is nested within a GPU implicit loop, which would be the case with e.g. matrix multiplication. It is important that the GPU explicit and implicit loops run for about the same number of iterations as all the GPU processing elements must run the same instruction. The cost of the GPU explicit loop is the same as it would have been on the CPU, except that there are fewer or even no data transfers.

CPU explicit loops These are the loops already present in the source code — loops run on the CPU, in serial. Both GPU implicit loops and GPU explicit loops can be written (safely) as a CPU explicit loop, however, this comes at the cost of performance as the CPU is only able to compute one iteration at a time.

3.5.1 Loop examples

GPU implicit loop Here is an example of a loop that could (trivially) be executed in parallel across the GPU:

```
for(var i = 0; i < 10000;i++){
  sum[i] = i*i+i;
}
```

Listing 33.0: A GPU implicit loop

GPU explicit loop We can extend the previous example to calculate which numbers are perfect numbers[1]:

```
for(var i = 1; i< 10000;i++){
  int divisors = 0;
  for(var j = 1; j <i; j++){
    if(i % j == 0)
      divisors += j;
  }
  if(divisors == i){
    sum[i] = i;
  }else{
    sum[i] = 0;
  }
}
```

Listing 33.1: A GPU explicit loop inside a GPU implicit loop

CPU explicit loop Finally, we may want to get the sum of all the perfect numbers we calculated:

```
//same code as before

int totalSum = 0;
for(int i = 0;i<10000;i++){
  totalSum= sum[i];
}
```

Listing 33.2: A CPU explicit loop

[1]A number is perfect if it is the sum of its proper divisors

Here we are unable to execute the code in neither a GPU implicit nor explicit loop as each iteration depends on those before it[1].

Extending Jikes (or any other virtual machine) does come with a number of advantages: it allows the user to speed up all of the programs targeted to that very virtual machine, including those already written. It leaves no further requirements with the programmer and it is a seamless experience for the developer. However, it does require that the user is aware of and install the extension, and the developer cannot assume that all users have the extension installed.

Instead of doing this, we will develop a plugin for the compiler. The resulting speed-up is limited to programs which are compiled with this extension, but it does not require the users to be aware of the existence of the extension, and the developers can count on it being available. In addition, it allows us to write the code at a higher conceptual level – source code instead of byte code.

3.6 MapReduce

MapReduce is both the name of a concept and software written by Google[DG04] which uses it to distribute computations over clusters of unreliable computers. The MapReduce software automatically ensures reliability and redundancy by splitting the computational task up over the different available computers.

MapReduce is based on the `map` and `reduce` functions from functional programming, and although Google originally used it scale across many unreliable machines, implementations of MapReduce have been created for multi-core[RRP+07] and GPUs[HFL+08, LCWM08, HCC+10].

[1]We will later show that this is possible with our system

4
Summary

We have investigated different techniques to program GPUs, including domain specific languages, functional programming as well as different ways to extend compilers to automatically use a GPU to increase execution speed.

We have also looked at MapReduce, the technique Google uses to distribute computations across computer clusters and which has been used in a number of different frameworks both for multi-core CPUs and for GPUs.

Going forward we will develop a plugin for the Scala compiler which will attempt to ofload map and reduce computations to the GPU without changing the meaning of the program or without requiring the programmer to learn any special syntax or API.

Part II

Design

5 The Scala Compiler Plugin System

Before we go into the technical details of the Scala compiler plugin, we provide an overview of the design of the plugin and the Scala compiler plugin system in general.

The overview follows the three requirements we have to the Scala compiler plugin:

1. It must update the AST to insert a call to our runtime system,
2. It must translate the map or reduce function argument to C so that it can run it on the GPU,
3. It must compute the benchmarks needed to decide at runtime if the code should run the GPU.

The documentation of the scala compiler and plugin system [Sca11a] [Sca11b] is not very comprehensive and we have thus examined the source for the 2.8.0 Scala compiler[1] which is the version we will use for this project.

The Scala compiler is large and uses many of the advanced features of Scala, but below is an overview of the most important components:

[1]The Scala compiler is undergoing rapid development, at the time we started to work on the project version 2.8.0 was the current version, at the time of writing 2.9.0.1 is the current version.

scala.reflect.generic.Trees This is the outer trait in which all the classes used to model the Scala abstract syntax tree exist. Confusingly enough, there is also a trait in the compiler called Trees (it is in package scala.tools.nsc.ast) which extends this trait.

scala.tools.nsc.plugins This package contains the PluginComponent and Plugin classes which our plugin must extend to work with the Scala plugin system.

Plugin This is the abstract class our plugin class must extend from. In order to do so, we must provide a bit of information, including the list of PluginComponent classes that the plugin consists of as well as the name of the plugin.

PluginComponent Each plugin consists of one or more PluginComponent classes, each of which describes exactly one compiler phase. This class contains the properties of the compiler phase, such as when it should run and how to handle commandline options.

scala.tools.nsc.transform.Transform This trait can be implemented by a PluginComponent class, should the component transform the abstract syntax tree. It will create a Phase which runs a Transformer.

Phase This is the class responsible for updating the abstract syntax tree.

Transformer This trait must be implemented by the class transforming the abstract syntax tree.

This might seem complicated but it allows us to make more complex plugins, adding features such as multiple inheritance which would require new parser, typechecker and code generator components as well as plugins which change the abstract syntax tree (so that the plugin works with the rest of the system).

To specify when the different plugin components should be called, we can specify which phases the compiler depends on and the Scala compiler will then arrange the plugins in a way so that all the requirements are satisfied.

5.1 AST Components

The AST[1] in the Scala compiler system is composed of case classes which all exists in the `scala.reflect.generic.Trees` trait. Each `PluginComponent` must provide either a `Phase` (which is used when we want to do additional checks on the AST, or to convert it into something else, such as Java class files) or, more common, a `Transformer` which is used to transform the AST itself.

Some of the most common AST nodes are:

Ident(name) is the node which represents an identifier.

Apply(tree, args) is node which denotes function application. `tree` is the function we wish to call and args is the list of arguments that it takes.

Select(tree, name) is the node which is used to select some value on an item. `tree` is the owner of what we wish to select, `name` is the name of what is wish to select on `tree`. `Select(Ident("horse"), "shoe")` is an AST fragment which represent the Scala code `horse.shoe`.

DefDef(modifiers, name, typeParameters, arguments, type, body) DefDef is the node that represent method definitions. `modifiers` are flags which indicate the protection level of the method, if it is overriding a method in a parent class, etc; `name` is the name of the function; `typeParameters` is the list of generic types that the method has – `map` has a generic type so that the return type can be adjusted depending on what argument it is called with; arguments is a list of lists of `ValDef`, in Scala functions can be curried such that not all their arguments are provided in which case they return a function which takes the rest of the arguments. To allow a function to be curried it has to be declared as `def a(i: int)(b: int)` – `arguments` is then a list of two list

[1]Abstract Syntax Tree

of `ValDef`; `type` is the type of the method and `body` is the body of the methods.

ValDef(modifiers, name, type, rhs) ValDef is the node that represent the definition of a value, either as a member of a method or class or as the part of the `arguments` to a method. Like `DefDef` it may have one or more modifiers, such as public or private; a `name` which is what it can be referred from in `Ident` nodes; a type and rhs which is the value it is initialized to when it is created.

The Scala statement:

```
println("Hello, world!")
```

Is parsed with an AST as:

```
Apply(
  Ident("println"),
  List(
    Literal(Constant("Hello, world!"))
  )
)
```

5.2 Phases

Previously we said that a compiler plugin could define one or more `Phases` through a `PluginComponent`. The Scala compiler also defines a number of phases:

parser	explicitouter
namer	erasure
packageobjects	lazyvals
typer	lambdalift
superaccessors	constructors
pickler	flatten
refchecks	mixin
selectiveanf	cleanup
liftcode	icode
selectivecps	inliner
uncurry	closelim
tailcalls	dce
specialize	jvm
	terminal

These phases matter since plugins must work on the AST as it is after the phases before it has been called. After the typer phase, the AST ofor the `println` statement looks like this:

```
Apply(
  Select(
    Select(
      This("scala"),
     "Predef"),
   "println"),
  List(
    Literal(Constant("Hello,␣world!"))
  )
)
```

For this reason, it is important that our compiler plugin is called the AST is rich enough that we can ensure only the correct code is instrumented.

6
Update the AST

Since the Scala compiler internally represents the program as an AST[1], we have to determine exactly which changes our compiler should make to the AST.

There are two cases where we should potentially change the AST:

map When it contains a `map` statement on a collection.

reduce When it contains a `reduce` statement on a collection.

However, not all instances of neither map nor reduce statements should be called, as it may either not be possible to run the code on the GPU (this is the case for code which has side effects, such as printing the value or updating a counter) or it may be possible, yet not desirable (this is the case, for example, if the collection is small and, due to moving it to the GPU and back, may take more time than just computing it on the CPU).

The simplest way to translate the AST would be to replace each call to map with code that copied the collection to the GPU, ran the computation on the GPU and copied the collection back to the main memory.

Thus, code which looks like:

```
collection.map(a=>a+a)
```

[1]abstract syntax tree.

would have to be translated to:

```
GPUAccelerate.map(collection, kernel)
```

where GPUAccelerate.map is a function that returns a collection.

While this approach is sound, it suffers from a number of performance issues:

- If the original map call is called on a collection with values that were already moved to the GPU, we copy the values back to the host computer just for them to be copied back on to the GPU immediately. It would be quicker to skip the extra copying.
- Without having the original function, GPUAccelerate is forced to run the code on the GPU even if the collection is very small.

The first issue can be solved by returning a special collection that the GPUAccelerate.map function can check for. The second issue can be handled by adding the function given to map as an additional argument to the GPUAccelerate.map function. This allows the map function to run and maintain the calculations on the CPU, should this be deemed to be better.

Likewise, then reduce call can be translated to:

```
GPUAccelerate.reduce(collection, kernel, function)
```

7
Translate from Scala to C

The reason for GPUAccelerate.map having to take a kernel argument is that whilst Scala does have first class functions we cannot use them as they rely on the JVM that is unavailable on the GPU. Instead, we will have to translate the anonymous function to C, which can be made to run on the GPU through the C compiler which is built into the OpenCL library.

As previously stated, however, it is not possible to translate all possible functions to run on the GPU, partially because we are incapable of providing any guarantees as to what happens with side-effects in the code (they may be called in any order and that particular order is not likely to be consistent in-between calls), partially due to things like object creation not being possible on the GPU.

Since this is a compiler extension, it should function in such a way that it does not change the meaning of the code, but merely the speed with which that code is executed. In general, it is impossible to mathematically determine whether two functions are equal—that they give the same result for all possible input values, that is—so we must be more conservative.

The compiler only considers moving the function to the GPU if the function given as the argument satisfies the following defined requirements:

- The function does not access any variable. This does not apply to values holding immutable objects (such as Javas String or int types), as they cannot be changed and so may safely be copied to the GPU.

- The function does not create any new objects.
- The only functions called are the arithmetic operators, as well as the functions available in the Java.math package with the exception of the random function.

These requirements all help ensuring that we only move functions whose execution does not change anything elsewhere.

The first limitation ensures that we do not access anything which may be changed by a different part of the program – this may result in the execution of the function being impure.

The second limitation is necessary as it is not possible to allocate memory from the GPU kernel and we cannot pre-allocate an area of memory and coordinate sharing it in-between the work-groups on the GPU, since our test hardware does not support device-wide atomic operations.

The final limitation is a broad one – it ensures that we for definite only call pure functions and that these are also available on the GPU. This limitation is not as large as it may come across as, at first, although: the major speed improvement on the GPU comes into force when it does computations on floating-point values, and most of the functions we would like to use with them are available in the java.math package.

There are many other functions in the Java standard library that are pure, but even so we cannot necessarily run them on the GPU. In Java, String objects are immutable, so all the functions in the String class are pure but we would need access to the source code for the String class in order to translate the functions to run on the GPU.

While we could write special code to handle String related functions, to translate arbitrary functions, we would have to prove that the function neither accesses anything outside its arguments nor calls a function which does and proves that this, too, is the case for all functions called, directly or indirectly, from this very function. This can only be done if all the called functions are on objects which we can either prove the dynamic type of (and, therefore which method actually implements the code) or where the called method is final or static (and, thus, cannot be overridden in a subclass).

Methods to automatically prove purity do exist, and they can be very efficient. A purity analysis system for Java is described in [Pea11] and is capable of proving the purity in large parts of the Java standard library; to do this, the author has to extend the concept of purity to include functions which create objects and access local, non-public properties. Unfortunately, we cannot create objects on the GPU, and as a result of this we are unable to utilise the extended purity system.

As a further result, we will limit the allowed functions in the map step to the aforementioned.

For the `reduce` function, we have to add some additional limitations as originally the function is either specified as `reduceLeft` or as texttttreduceRight, both of which guarantee the order in which the reductions are executed. Since no such guarantee is possible on the GPU, we have to add the additional requirement that the function given is associative. Regrettably, it is not possible to mathematically decide whether a function is associative, so we will only translate functions for reduction if they also satisfy the requirements listed below:

- The input type is either a float or an integer.
- No non-associate function is called on the arguments.

Both of these requirements are vital to prevent the situation where the if-statement can be used to create a reduce function which is not associative. If we allowed the input value to be of a more complex type, we could define the reduce function to evaluate

```
if(a.someVal > b.someVal){
  a
  }else{
  b
}
```

This will ensure that the value returned from the reduce call is the one with the largest someVal – that is, if no other object in the input collection has a someVal which is greater – however, if two objects both have the largest value, then the one being returned depends on in which order the input is given in.

If we had left out the second requirement, one could write the function to reduce (with arguments a and b)

```
a−b
```

In this case, the result also depends on the order in which the arguments are evaluated by the function.

A minor problem occurs, however – due to rounding errors, floating point addition and multiplication are not associative. Unfortunately most of the computations we can expect to do, and where the GPU is really swift, are on floating point math. Fortunately, while we cannot provide the same result as if the code had run sequentially, the result is still correct. We therefore choose to ignore this minor problem.

Once we have determined whether to run a function on the GPU, we have to translate it. For simple examples like:

```
a => a + a
```

this is easy, and since all the math related functions in Java.math have a corresponding C function, we can simply substitute these as needed.

Translating reduce calls are more complicated because they depend on the value of the result of a previous calculation.

In order to ensure that the work items have access to the previously computed values, we have to compute the reduced values over a series of rounds, where, for each round, each active work item computes one reduced value (from two input values) and then, after synchronising with the rest of the work group on a barrier, every other active work item becomes inactive. We continue doing this until there is only one value left for each work group. Since we cannot synchronise in-between work-groups, we have to either copy the result back to the main memory and reduce it there or run a new kernel which computes the final

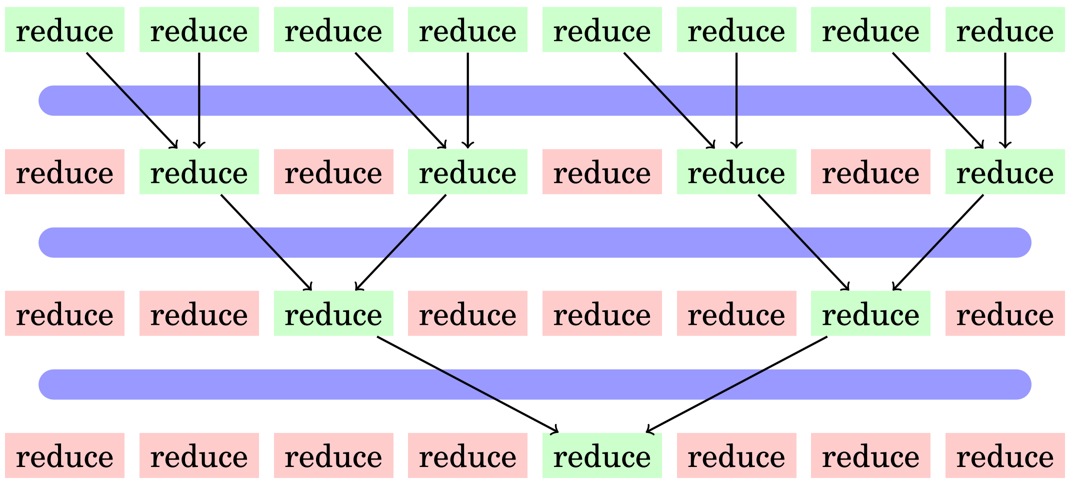


Figure 51.1: Diagram showing the data-flow between the work-item computations. Red nodes indicate inactive compute units, green indicate active compute units and the blue lines are where the barriers are inserted.

reduced value – the process for a single work-group is illustrated in Figure 51.1.

Take the number of vehicles in the field, (A), and multiply it by the probable rate of failure, (B), then multiply the result by the average out-of-court settlement, (C). A times B times C equals X...
If X is less than the cost of a recall, we don't do one.
Jack, Fight Club

8 When to Run code on the GPU and When not to

Just because we can run the code on the GPU does not mean we have to. If the code runs on the CPU, we do not have to initialise OpenCL, copy the values to the GPU, schedule and execute a kernel nor copy the results back. If the input is small enough, the time saved can be significant.

The interesting question is now: How do we estimate the cost, i.e. the required time, to do the computation on the GPU versus the CPU?

One way to determine the expected cost of running the code on the GPU is to consider the size of the input and resulting output data, which have to be copied to and from the GPU, and divide this by the bandwidth between the GPU and the CPU; then, add the expected time required to perform the actual computation. While it may be possible to get a reasonably accurate estimate of the transfer time, the code sent to the GPU is optimised by a JIT compiler, so there is no way of knowing exactly what code with be executed and we have not found any public technical documents which tell how much the various operations cost anyway so we would not be able to calculate this, even if we knew exactly what operations where done.

The alternative approach to this is to benchmark the compiled kernel on a few values and then time this takes decided whether or not use the GPU. Unfortunately this would require each kernel – and there may be many kernels in a given program – to be tested which would result in a significant delay.

A compromise would be to test some basic operations and then assume than the execution speed of the resulting program is about the same as the execution time of the parts it consist of. As the kernels are likely to be relatively simple, this would be a reasonable approximation.

Then we could benchmark the same operations on the CPU, compare the numbers and see which one would be the fastest; benchmarking the CPU would still take time, however.

Instead of doing this, we can get a very rough estimate by dividing the number of GFLOPS the GPU can produce with the number of GFLOPS the CPU can provide. In the case of our test equipment, that is about 8.75. If we further assume that because of overhead and underused GPU-cores we only get a bit less than half of that benefit, this means that our GPU is about 4 times faster than the CPU.

Our test machine can run (on the CPU) 2 billion iterations of a for loop with a floating-point multiplication through in about 2.14 seconds and takes about 39.5 seconds to do the same to a for loop with a call to the “sin” function[1]. If we assume all the allowed functions are equally expensive, we can construct the estimated time-cost table:

	CPU	GPU
function call (sin, cos, etc)	39.50	9.9
arithmetic or logic operator (*, +, -, /, >, =, etc)	2.14	0.535

Of course these numbers will differ depending on what hardware the software runs on, but they will be accurate enough to prevent us from sending very small arrays to the GPU, or avoid sending very large ones.

Now we just need a way to compute the estimated time a program will take. To do that, we will use a slightly modified WCET-analysis.

[1]This include the time to start the program with the “time” unix command, but that is insignificant compared to the total running time.

WCET-analysis WCET[1] as a method used in real-time programming to determine the most time a given program can take to complete. We can use the same technique, although we need not concern ourselves with the worst case, but rather the average case.

The basic idea behind WCET-analysis is to add the time each operation in a program takes, the sum of which is the worst case execution time. This is easy enough for simple operations, but for loops we need to know the maximum number of times the loop may be run through, which is often not possible to know.

With the restrictions we put on the programs we can move to the GPU however, it is trivial. First we can expect most of the functions to be simple, and if they contain loops then the number of iterations of these loops must either be constant, depend on a value on the input variable or on some other value available in the current environment (since we do not translate functions which refer to variables) in which case we can check it before we determine whether to move the computation to the GPU or not.

Finally we need some measure of the cost of sending data to the GPU. The Cuda SDK bandwidthTest sample says that the bandwidth from main memory to the GPU is 1428 MB/S, and from the GPU to main memory is 1154 MB/S on our test computer. This is fast enough that the limitation is mainly going to be latency for most cases. Looking at the graphs in [Hov08, p. 17 - 25], we can expect read and write latencies of between 8 and 12 microsecond for all the tested cards. 8 microsecond is about the time it takes to do 400 iterations of the floating point multiplication loop.

[1]Worst-Case Execution Time

Part III

Implementation

9
Implementation

To implement the compiler, we will use the plugin framework included with the Scala compiler[1]. This framework allows us to access and change the AST during the compilation, after any phase we choose.

The compiler plugin will run after the "‘refchecks"’ phase so that it has access to type information about the elements in the tree.

The compiler, as outlined in the design part, has three tasks:

1. Update the AST, replacing map and reduce calls with calls to our runtime.
2. Translate the function given with the map call into OpenCL C code.
3. Analyze the function and extract the information necessary to estimate whether it would be a benefit to run it on the GPU.

9.1 Update the AST

As said before, the compiler plugin framework gives us direct access to AST so the translation is reasonably simple. All we have to do is find all instances of

[1]Version 1.8.0

```
floatCollection.map(function)
```

with our runtime code:

```
GPUAccelerator.mapf(floatCollection, function,
   ↪     functionCcode, benchmarkInfo)
```

In Scalas AST this can represented as

```
Apply(Select(TermName(GPUAccelerator),TermName(mapf)),
   ↪      List(floatCollection, function,
   ↪     compileToOpenCL(function), benchmarkFunction(
   ↪     function)))
```

Where compileToOpenCL is the function which compiles the AST to C code and benchmarkFunction is the function which computes the estimated cost of running the code on the GPU versus CPU.

The first thing to do is to search through the Scala AST to see if we have some part of the AST which can be run on the GPU. We do this with Scalas pattern-matching capabilities:

```
def matchMapCall(potential: Tree) = {
  potential match {
    case s @ Apply( ta @ TypeApply(Select( Apply(
       ↪     Select(Select(This(name:Name), predef:
       ↪     Name),doubleArrayOps:Name),
       ↪     sourceCollection), mapCall:Name),typeTree
       ↪     ), functionArg) if ((name.toString == "
       ↪     scala") && (predef.toString == "Predef")
       ↪     && (doubleArrayOps.toString == "
       ↪     doubleArrayOps") && (mapCall.toString ==
       ↪     "map") &&(sourceCollection.length==1) &&
       ↪     (sourceCollection(0).isInstanceOf[Ident])
       ↪     ) => {
    true
    }
    case default => {
      false
    }
  }
}
```

It looks more complicated than it is – what we do is first to match on the general structure of the code, then we check that the function called is “map”, that we are calling it on the doubleArrayOps (a wrapper class that allows a high level view of Javas arrays, in this case with doubles), that the given function takes exactly one argument.

The Scala plugin system gives us access to the AST, but there is no way to iterate through all the nodes in the AST. Instead we walk the tree and for each node which may be the outer-most node in a map call or contain a map-node, we run through its child nodes. This is all done in the replaceMapCall function:

```

def replaceMapCall(potential: Tree) : Tree = {
  potential match {
    case s @ Select(lhs, rhs) => {
      treeCopy.Select(s, replaceMapCall(lhs), rhs)
    }
    case s @ Block(l, e) => {
      treeCopy.Block(s, l, replaceMapCall(e))
    }
    case s @ DefDef(a,b,c,d,e, exp) => {
      treeCopy.DefDef(s,a,b,c,d,e, exp)
    }
    case s @ Assign(l, r) => {
      if (matchMapCall(r)) {
        return treeCopy.Assign(s,l, translateMapToGPU(
            ↪    r))
      }
      treeCopy.Assign(s, l, replaceMapCall(r))
    }
    case s @ Apply(f,rhs) => {
      treeCopy.Apply(s,
      if(matchMapCall(f)){
        translateMapToGPU(f)
      } else {
        replaceMapCall(f)
      },
        rhs.map{a => if(matchMapCall(a)){
            translateMapToGPU(a)
          } else {
            replaceMapCall(f)
        }})
      }
      case d @ default => {
        super.transform(tree)
      }
    }
}
```

Unfortunately this code attaches an extra element to the tree each time it is called. Since the documentation for the Scala plugin system is scattered in many different places and other Scala compiler plugins we looked at did not seem to have this problem, we decided to skip the automatic translation part and simply insert the code manually when we benchmark the system.

Since it is possible to write a plugin to the Scala compiler, we are more interested in how well the compiler plugin would optimize the code if we fixed the issue.

9.2 Translation into C

Since we have very strict limits on what the function arguments can do, it is not very difficult to translate them into C code. All that is required is that we change the AST from:

```
Apply(Select(value, functionName), arg)
```

to

```
value functionName arg
```

or if functionName is not an arithmetic operator[1] then translate it into

[1]An arithmetic e.g *,/,-,+

```
functionName(value)
```

The translation for the reduce function is essentially the same, except that - and * cannot be used as operators because they are not associative.

Because we cannot get the compiler plugin to work, we will not implement this part since it is not needed.

9.3 Data to Benchmark

To avoid the cost of moving small arrays back and forth between the GPU and CPU, we implement a simple benchmark function which computes the estimated cost of running the code on the GPU. We use the price found in the design section, rounded up to the nearest integer.

```
def benchmark(tr : Tree) : int = {
            def costOf(f: String) => {
                  if(f.length > 2)
                    10
                  else
                    1
                }
                tr match {
              case t @ Apply(a, args) =>
                    args.map(benchmark).foldLeft(0)(_+_)+ benchmark(a)
                  case Select(a, name) =>
                    benchmark(a) + costOf(name.toString)
                  case If(a, b, c) =>
                    benchmark(a) + benchmark(b)+ benchmark(c)
                  case default => 0
            }
          }
```

There are two things that should be noted here: First the reason we check the length of the functions name is that all the arithmetic operators all have names with a length two or less whereas all the functions from `Java.math` have a name with length at least three.

The second thing to note is that we compute the price of the if statement as the cost of both branches rather than the average cost. This is because on a GPU the cores run in lock step so if one core branch then the others will follow along (though they will not modify any data). Since the CPU is not limited in this way the function may estimate the cost of running on the CPU a bit too high (the cost to run on the CPU is given in the design section as 4 times the cost of the GPU for the same operation).

To decide if we should run it on the GPU all we have to do is to add the time it takes to move the data to the GPU to the time it takes to run it and if that is lower than the cost of running the code on the CPU, move it.

9.4 The runtime

To run code on the GPU, we will use the open source library JavaCL[1].

We would need an object with two methods, one for map and one for reduce, however the benchmarks which we are able to optimize only requires map, so we will only use that.

Because of some weird bugs in the way Scala interacts with the JavaCL system, the part our code which is responsible for calling through to OpenCL is written in Java.

This is the Java part, based heavily on the code which drives the OpenCl benchmark suite:

```
package Runtime;

import com.nativelibs4java.opencl.*;
```

[1]http://code.google.com/p/javacl/

```java
import com.nativelibs4java.opencl.util.*;
import com.nativelibs4java.util.*;
import org.bridj.Pointer;
import static org.bridj.Pointer.*;
import static java.lang.Math.*;

public class JRuntime {

  static CLContext context;
  static CLQueue queue;
  static Pointer<Float> aPtr;
  static CLBuffer<Float> a;
  static CLBuffer<Float> out;
  static CLProgram program;
  static CLKernel kernel;
  static CLEvent evt;
  static Pointer<Float> outPtr;

  public static float[] mapf(String src, float[] input
      ) {
    context = JavaCL.createBestContext();
    queue = context.createDefaultQueue();
    aPtr = Pointer.allocateFloats(input.length);
    for (int i = 0; i < input.length; i++) {
      aPtr.set(i, input[i]);
    }
    // Input buffer
    a = context.createFloatBuffer(CLMem.Usage.Input,
        aPtr, true);
    // Output buffer
    out = context.createFloatBuffer(CLMem.Usage.Output
        , input.length);
    // Compile program source
    program = context.createProgram(src);
    kernel = program.createKernel("mapf");
    kernel.setArgs(a, out, (float)input.length);
    evt = kernel.enqueueNDRange(queue, new int[] {
        input.length });
    outPtr = out.read(queue, evt);
    float[] o = new float[input.length];
    for(int i = 0; i < input.length; i++) {
            o[i] = outPtr.get(i)
        }
        return o;
    }
}
```

The Scala program will then have to look:

```
object GPUAccelerator {
  mapf(collection : Array[Float], fun : Float => Float, openClCode: String, benchmarkInfo: Int) : Array[Float] = {
    //We assume that if it will take more time to run than the latency, it is worth it.
    //Latency is for both ways assuming 10 microseconds each
    if((benchmarkInfo * collection.length)/(2000000000.asInstanceOf[Double]) > 0.00002){
      Runtime.JRuntime.mapf(openClCode, collection)
    }else {
      collection.map(fun)
    }
  }
```

Part IV

Performance

10 Benchmarks

There are many different benchmarks used in high performance computing and authors often create their own benchmarks as well. Additionally some benchmarks require fairly domain specific knowledge, e.g. the BioPerf framework which uses algorithms such as phylogenetic reconstruction and protein structure prediction[BLLS06]. In [BLWJ], we identified a number of benchmarks:

Linpack This is the benchmark used to determine the TOP500 rankings of super computers. While this is a simple benchmark, all it does is test how fast a cluster can do floating point calculations — an area where we already know the GPU excels.

NAS benchmark This is a more general benchmark, which is based on problems encountered in fluid-dynamics. Unfortunately half of the problems test communication between the nodes in the cluster (which is not an issue when the HPC consist of one node), so that benchmark is not that useful in this case.

hiCUDA benchmark These benchmarks are from the paper on hiCUDA and are according to the authors standard CUDA benchmarks[HA09]. They include matrix multiplication, sum of absolute differences as well as a polynomial equation solver.

Since we are not trying to compare our solution directly to that in the paper on hiCUDA, we can afford to relax their benchmarks, as well as to add those from the NAS benchmark that test additional things. In [BLWJ] we found the following benchmarks:

- Solving a simulation of interacting points.
- Evaluating a function over an area.
- Solving a polynomial equation.

However it is not enough to decide which benchmarks to use, we must also find a way to evaluate them with a minimum of interference. [GBE07] suggest among other things that we consider:

1. One hardware platform versus multiple hardware platforms.
2. One heap size versus multiple heap sizes.
3. A single VM implementation versus mulitple VM implementations.
4. Back-to-back measurements versus interleaved measurements.

We can however remove a few of them. Since we are testing Scala and it is almost always used with the Java Virtual Machine from Sun we do not have to consider other implementations of the Java Virtual Machine.

In addition Scala is most often invoked with the standard size for the Java heap, so we will only test that. [BMG$^+$08] does have an example of the danger of doing so, it shows a test with two different garbage collectors where the fastest depends a great deal on the heap size. But this is not a concern if it is mostly run at a given heapsize – because we want the system to be as fast as possible for that particular heap size.

With regards to the hardware platform, we have only one machine to run benchmarks on, so although this may influence the final result we will not do the benchmarks on different hardware platforms.

Both [GBE07] and [BMG$^+$08] also point out that we need to control for non-determinism. This is an extra large issue in our case, because in addition to the normal issues we would have with the JVM JIT compiler, we also have a JIT compiler that runs on the GPU.

There are a number of techniques to remove or lower the non-determinism from the JIT compilers. [GBE07, BMG+08] suggest that we either focus on the warm-up phase of the JIT compiler – this is useful for benchmarks which takes only a short time and whose execution time is therefore greatly affected by the time it takes to start the system, load classes and perform the initial JIT compilation – or on the steady state, which is where the largest cost is incurred for benchmarks which takes a long time.

Since it takes some time to send the data to the GPU, initialize the kernel and read the result back it would not make sense to do this for short workloads so we will be writing the benchmarks to test the steady state of the program.

To make the benchmark tests we will run the tests multiple times in the same VM instance as suggested in [GBE07] (this allows the virtual machine to use the JIT compilers optimization) as well as running the same benchmark a number of times (to smooth out the results of the non-determinism from the JVM JIT compiler).

Unlike normal benchmarks for managed runtimes, we also have to consider the costs of the JIT compiler on the GPU. While we could certainly test the steady state of the JIT compiler on the GPU as well, that would be wrong as it would make the GPU appear faster than it really is and we would not be able to accurately compare the GPU optimization with the non-optimized Scala code. For this reason, the benchmarks have to include the entire cost of doing the computation on the GPU, including both the time to copy the data back and forth as well as the cost of the JIT compiler.

Finally then, we have to decided what benchmarks to run. Keeping in mind the limitations of our compiler and the list of benchmarks we got from the last report [BLWJ], we will be using the following benchmarks:

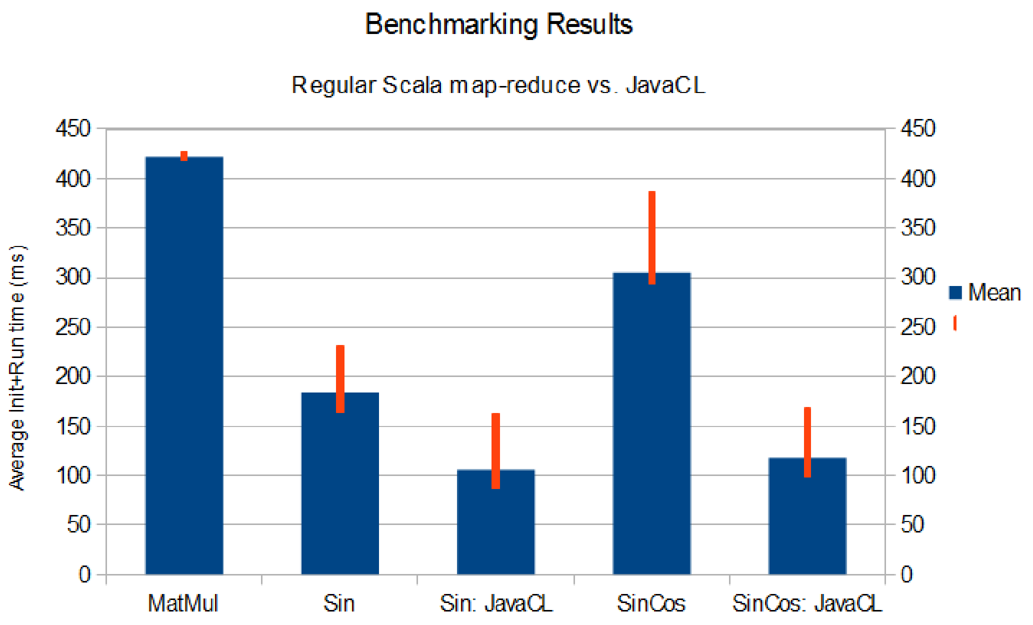
1. Matrix multiplication
2. $\sqrt{\sin(x) + \cos(x)}$
3. $sin(x)$

We chose Matrix multiplication because it is often used in high performance computing and illustrates the limit of our compiler plugin, since it is unable to optimize it.

We chose the $\sqrt{\sin(x) + \cos(x)}$ function because it is simple but have a high workload-data ratio.

We chose the $sin(x)$ function because it has a lower workload-data ratio.

Our test hardware contains a quad-core Intel Xeon E5420 processor and two NVIDIA Tesla C870 GPUs, but we will only be using one of the GPUs. The Xeon can deliver no more than 40 GFLOPS, while each Tesla card can deliver up to 350 GFLOPS performance.



Part V

Conclusion

"what 's done is done".

Macbeth (Act III, Scene II)

11 Conclusion

We described how to make a compiler plugin which can optimize programs by moving computations to the GPU. Unfortunately this compiler plugin cannot correctly update the AST of the input data, so the only way to use it is to insert the calls to it direcly in the source code, neglecting the benefit of the compiler plugin.

We ran some benchmarks comparing the GPU and Scala code, but we did not do any benchmarks directly on our plugin, partly because the GPU code would be identical to the GPU code we wrote and partly because of the issues we mentioned with the compiler.

We did find that significant speed-ups could be made by using a GPU.

If you can look into the seeds of time, And say which grain will grow, and which will not, Speak.
Macbeth, Act I, scene iii

12 Future work

Most importantly, we want the compiler to correctly update the AST, since the idea with the plugin is to make it so that the programmer does not have to worry about using the GPU.

Plenty of extensions can be made to the compiler plugin. In particular we would like to be able to ofload computation on not just arrays but also ranges, lists, and Scalas other standard collections.

In additions we would like to have the compiler do work on datatypes other than floats and integers, including case classes, tuples and strings.

We would also like a better heuristics for when it is advantages to move code to the GPU, as well as a more comprehensive benchmark so that we can better know the likely speed-up.

In particular however, we would like to be able to inline loops, calls to map and reduce and have a proactive caching system to move data to and from the GPU.

Bibliography

[amd] *Introduction to opencl programming*, http://developer.amd.com/zones/OpenCLZone/courses/Documents/Introduction_to_OpenCL_Programming%20Training_Guide%20(201005).pdf.

[Bit11] Bitcoin Wiki, *Why are amd gpus faster than nvidia gpus?*, https://en.bitcoin.it/wiki/Why_a_GPU_mines_faster_than_a_CPU#Why_are_AMD_GPUs_faster_than_Nvidia_GPUs?, 5 11.

[BLLS06] David A. Bader, Yue Li, Tao Li, and Vipin Sachdeva, *Bioperf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications*, Tech. report, Georgia Institute of Technology, 2006.

[BLWJ] Reidar BecK, Helge Larsen W., and Tommy Jensen, *Democratizing general purpose gpu programming through opencl and scala*, Available through AAU project library.

[BMG$^+$08] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann, *Wake up and smell the coffee: evaluation methodology for the 21st century*, Commun. ACM **51** (2008), 83–89.

[Chu41] A. Church, *The calculi of lambda conversion*, Princeton University Press, 1941.

[Coo10] Andrew Cooke, *A practical introduction to opencl*, http://www.acooke.org/cute/APractical0.html, 12 2010.

[DG04] Jeffrey Dean and Sanjay Ghemawat, *Mapreduce: Simplified data processing on large clusters*, Tech. report, Google, Inc, 2004.

[fas11] fastcompany.com, *1,000 core cpu achieved: Your future desktop will be a supercomputer*, `http://www.fastcompany.com/1714174/1000-core-cpu-achieved-your-future-desktop-will-be-a-supe` 1 2011.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout, *Statistically rigorous java performance evaluation*, Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (New York, NY, USA), OOPSLA '07, ACM, 2007, pp. 57–76.

[HA09] Tianyi David Han and Tarek s Abdelrahman, *hicuda: high-level directive-based language for gpu programming*, Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, 2009.

[Has11] Haskell, *Functional programming*, `http://www.haskell.org/haskellwiki/Functional_programming`, 07 2011.

[HCC+10] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin, *MapCG: writing parallel program portable between CPU and GPU*, Proceedings of the 19th international conference on Parallel architectures and compilation techniques (New York, NY, USA), PACT '10, ACM, 2010, pp. 217–226.

[HFL+08] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang, *Mars: a mapreduce framework on graphics processors*, Proceedings of the 17th international conference on Parallel architectures and compilation techniques (New York, NY, USA), PACT '08, ACM, 2008, pp. 260–269.

[Hov08] Rune Johan Hovland, *Latency and Bandwidth Impact on GPU-systems*, Tech. report, Norwegian University of Science and Technology, 2008.

[IBM11a] IBM, *Jikes*, `http://jikes.sourceforge.net/`, 2 11.

[IBM11b] ______, *Jikesrvm*, `http://jikesrvm.org`, 2 11.

[Int07] Intel, *Multicore: The software view - intel*, `http://download.intel.com/corporate/education/emea/event/af12/files/cownie.pdf`, 2 2007.

[LCWM08] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng, *Merge: a programming model for heterogeneous multi-core systems*, Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (New York, NY, USA), ASPLOS XIII, ACM, 2008, pp. 287–296.

[LLL09] Alan Leung, Ondřej Lhoták, and Ghulam Lashari, *Automatic parallelization for graphics processing units*, Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (New York, NY, USA), PPPJ '09, ACM, 2009, pp. 91–100.

[Moo65] Gordon E. Moore, *Cramming more components onto integrated circuits*, Electronics **38** (1965), no. 8.

[Oli11] Olivier Chafik, *Scalacl*, `http://code.google.com/p/scalacl`, 06 2011.

[ope] *The opencl specification*, `http://www.khronos.org/registry/cl/`.

[Pea11] David Pearce, *Jpure: A modular purity system for java*, Compiler Construction (Jens Knoop, ed.), Lecture Notes in Computer Science, vol. 6601, Springer Berlin / Heidelberg, 2011, 10.1007/978-3-642-19861-8_7, pp. 104–123.

[RRP$^+$07] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis, *Evaluating MapReduce for multi-core and multiprocessor systems*, In HPCA '07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture, IEEE Computer Society, 2007, pp. 13–24.

[Sca11a] Scala, *Scala developer guides*, `http://www.scala-lang.org/node/215`, 07 2011.

[Sca11b] ———, *Writing scala compiler plugins*, `http://www.scala-lang.org/node/140`, 07 2011.

[Sca11c] Scala Reviewer, *Optimize simple for loops*, `https://issues.scala-lang.org/browse/SI-1338`, 06 2011.

[Sut05] Herb Sutter, *The free lunch is over: A fundamental turn toward concurrency in software*, Dr. Dobb's Journal **30** (2005), no. 3, 202–210.

[TIO11] TIOBE Software, *Tiobe programming community index*, `http://www.tiobe.com`, 07 2011.