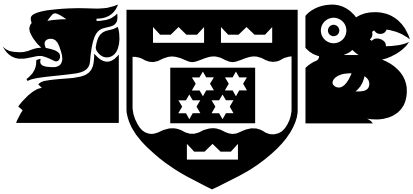
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Compiling Whiley Programs for a General Purpose GPU

Melby Ruarus

Supervisors: Dr. David Pearce, A/Prof. Lindsay Groves, Roman Klapaukh

Submitted in partial fulfilment of the requirements for Bachelor of Engineering with Honours in Software Engineering.

Abstract

This project investigates improving the performance of Whiley programs by executing portions of these programs on GPUs while maintaining as closely as possible the semantics of the language. Programs written in languages such as Whiley are typically not well suited to execution on GPUs which exhibit large-scale data parallelism in contrast to small-scale task parallelism seen on CPUs. Therefore, the developed solution parallelises only the portions of programs which are likely to benefit—specifically, certain types of loops—and applies several optimisations to increase performance. The evaluation of this technique validates the approach with one benchmark exhibiting a 5.2x speed improvement, and identifies areas where the compiler can produce further improved code.

Acknowledgments

I wish to foremost thank my supervisors, David Pearce, Lindsay Groves and Roman Klapaukh who have put so much time into providing feedback, advice, and for helping make this project a reality.

To all those who have helped proofread this report, every bit makes a big difference, I am very grateful for your help.

I would also like to recognise the School of Engineering and Computer Science staff who have provided me with the resources I needed to perform my evaluation, the help is greatly appreciated

Contents

Figures

Tables

Chapter 1

Introduction

Programming for Graphics Processing Units (GPUs) is very different from Central Processing Units (CPUs) due to differences in hardware architecture. This difference means that programs written for a CPU cannot be simply converted to take full advantage of a GPU. Traditionally different programming languages, methodologies and skills are required to write GPU programs, meaning that developers need expertise in a specialist field.

The aim of this project is to speed up the execution of any Whiley program by making use of the hardware available on modern computers. This is achieved by transparently converting portions of programs written for the CPU to be run on a GPU—providing a speedup without any additional work from the developer.

Whiley is a hybrid object-oriented and functional programming language [36, 37] which has several language features that both help and hinder conversion to a GPU. While the powerful theorem prover which is part of this language is not utilised in this project, it offers many future opportunities for improving the analysis required to determine which sections of programs can be run on a GPU.

The three formal requirements for the project are as follows:

1. programs compiled to run on graphics hardware exhibit improved performance,

2. the behaviour of the Whiley programming language is maintained as closely as possible,

3. the programmer need not write additional code or be aware of the fact that portions of the program will run on the graphics card.

1.1 Contributions

By addressing these requirements this project has produced three key contributions:

- the development of a compiler which is capable of compiling Whiley programs of which portions run on a GPU,

- the development of optimisations to improve the performance of these programs when running on GPUs,

- an evaluation of the performance of these programs and the impacts of these optimisations which validates the approach taken by this project.

1.2 Organisation

This report is structured starting with the background in Chapter 2 which covers the differences between CPUs and GPUs, the architecture of GPUs, how programs are written to execute on these, the Whiley programming language and other work related to this project. This is followed in Chapters 3 and 4 by the design of the solution and details of the architecture and basic implementation. Chapters 5 and 6 go into further detail on two specific aspects of the solution, selecting which parts of the program to run on the GPU, and performing optimisations to improve the performance of the code. An evaluation of the solution is then performed in Chapter 7 to investigate how closely the solution addresses the first and third requirement, followed by Chapter 8 on future directions for this project. Chapter 9 presents this reports conclusions.

Chapter 2

Background

In recent years hardware designers have increasingly turned to systems with large degrees of parallelism to improve performance. However developing software to exploit these new architectures requires additional work by the developer, both to understand the different ways in which these devices behave, and to integrate the variety of programming languages used.

2.1 GPUs vs CPUs

The fundamental difference between CPUs and GPUs lies in the form of concurrency they offer. CPUs are **task-parallel** whereas GPUs are **data-parallel** (Figure 2.1).

2.1.1 Task Parallelism

Each core in a CPU is a fully functioning processor, able to execute different instructions on different data independently of any other cores on the chip. For this reason this is called task-parallel where every core can be performing a different task—otherwise known as Multiple Instruction Multiple Data (MIMD). Each core on a CPU has its own Arithmetic Logic Units (ALUs), cache, registers, program counter, etc. For this reason each CPU core is very complex, essentially being a different CPU in the traditional sense.

2.1.2 Data Parallelism

Each core in a GPU executes the same instructions on different data as cores share the same program counter and caches while having their own ALUs and registers. For this reason, each thread of execution is performing the same computation, only with different data to operate on. This architecture is called data-parallel, and these processors are also known as Single Instruction Multiple Data (SIMD) devices.

2.1.3 How These Compare

The GPU SIMD model is less general than MIMD as used in CPUs, though it is significantly simpler to build, allowing manufactures to produce many more cores than can be done with a CPU. This means a typical GPU will have hundreds of cores, while a modern CPU will have between two and four.

While GPUs contain hundreds more cores than CPUs, this does not necessarily mean they are hundreds of times faster. Due to the different nature of these processors, a problem must suit the architecture of a GPU for it to benefit from the greater number of cores. GPU

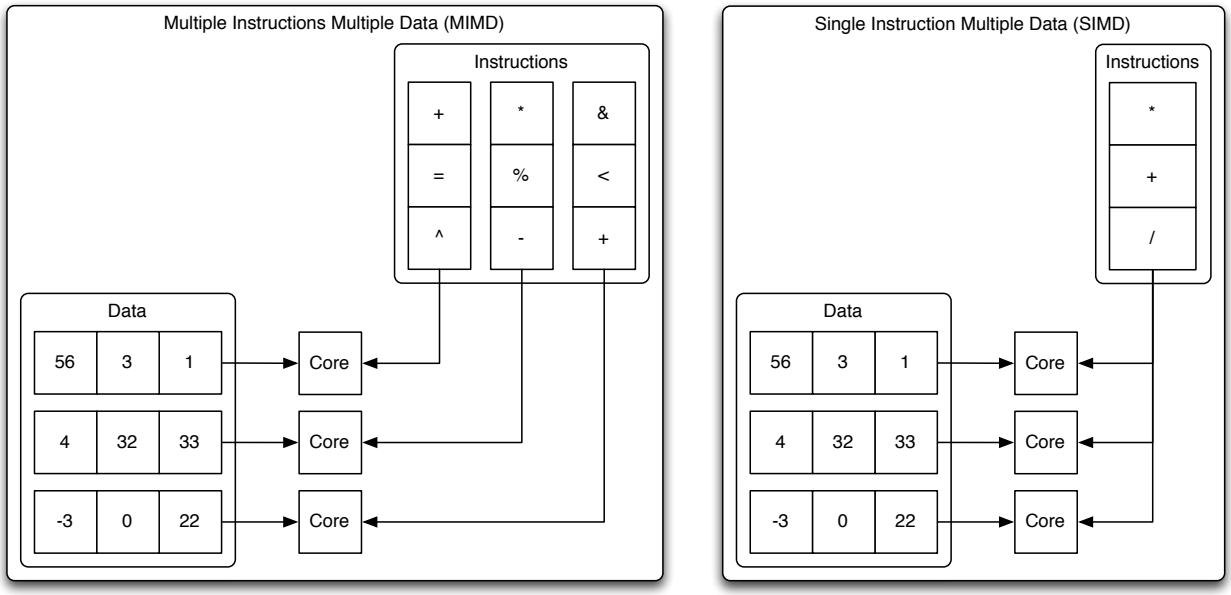


Figure 2.1: Task-parallel Multiple Instruction Multiple Data (MIMD) vs data-parallel Single Instruction Multiple Data (SIMD).

cores are typically lower-clocked (slower) than those in CPUs, but due to the sheer number of these processors GPUs can provide more raw computational power when utilised effectively [35, 28].

2.1.4 Why This Matters

Modern programming languages and frameworks are typically developed to run on CPUs. Because of this they are often ill suited to utilise the data parallelism afforded on data-parallel devices as they assume one thread of execution per task. Further complicating this issue is the different way in which program execution must be considered by the developer and the fact that certain computational problems are naturally better suited to data-parallelisation than others [30].

Solving this problem for some tasks utilising GPUs can provide significant performance improvements, in some cases hundreds of times faster than execution on a CPU [14].

2.2 GPU Architecture

The task GPUs were originally constructed for was accelerating the rendering of 2D and 3D graphics, a problem which lends itself well to parallelisation. Traditionally these devices were entirely dedicated to rendering, and it was not possible for software developers to make any use of this hardware. In the beginning of the 2000s programmable shaders were introduced in Open Graphics Library (OpenGL) [26] and Direct3D [7], allowing graphics pipelines to be extended with custom programs for geometry or shading calculations. These changes opened up GPUs to be used for more general computational problems, and more recently the development of general purpose programming Application Programming Interfaces (APIs) such as Open Compute Language (OpenCL) [24] and Compute Unified Device Architecture (CUDA) [32] have made General-Purpose Graphics Processing Unit (GPGPU) programming accessible to many.

NVIDA and AMD are the leading manufacturers of GPU hardware, producing a range of different products aimed at graphics processing, and some specifically designed for GPGPU computation. While the high level architectures of these products are largely the same, NVIDIA and AMD use different low level hardware, details of which remain trade secrets. Even across different generations of products from the same vendors performance characteristics differ, though the general trend from AMD has been for more, simpler ALUs with lower clock frequencies compared to NVIDIA who use fewer, but more complex ALUs. GPGPU programming APIs have been developed to largely shield programmers from these details, though some knowledge of these differences is needed to effectively make use of the underlying hardware.

2.3 OpenCL

For this project OpenCL has been selected as the primary GPGPU programming API due to its wide support for different hardware [14]. Programs using OpenCL can utilise GPU hardware from both AMD and NVIDIA, CPUs from the likes of AMD and Intel and less common hardware such as Field Programmable Gate Arrays (FPGAs) [25].

OpenCL was initially developed by Apple and released with Mac OS X Snow Leopard with the goal of creating a cross platform system for GPGPU computation. It has since become an open standard, and OpenCL 1.1 was ratified by the Khronos Group on 14 June 2010. At the writing of this report, the current version of OpenCL is 1.2, which was finalised on 15 November 2011 [24].

A C like programming language is used in OpenCL for writing programs, in conjunction with an API for controlling execution and managing resources. OpenCL can be described in the context of a hierarchy of models, these are: the platform model, the execution model and the memory model [25].

2.3.1 The Platform Model

OpenCL abstracts the physical hardware architecture of the host machine and the devices upon which computation will be performed. In this model the host is connected to one or more compute devices, of which the host may be one. These compute devices are divided into one or more compute units, which are further divided into processing elements which actually perform computation [25] (see Figure 2.2).

2.3.2 The Execution Model

OpenCL programs execute in two parts, kernels that execute on compute devices and a program which runs on the host defining the context within which kernels run, and managing their execution.

The core of the OpenCL execution model is the kernel. A kernel is a short program which is dispatched for execution over a range of input values. An instance of this kernel will then execute for every item in this range. A particular instance of the kernel run for a particular input value is referred to as a work-item. The index of the work-item into the input range provides a global ID for the work-item.

Several work-items are grouped together into a work-group, within which each work-item has a local ID. Work-items within a work-group will run concurrently, and will be executed on the same compute unit (see Figure 2.3).

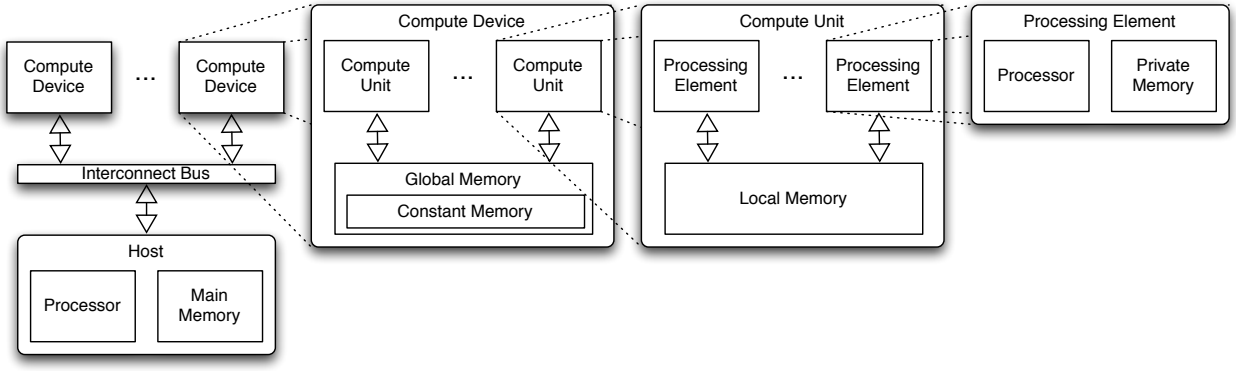


Figure 2.2: The OpenCL platform model. The host device is connected to one or more compute devices (of which it may be one) over an interconnect bus, Each of these devices has a large region of global memory, and one or more compute units. Compute units (in many cases) have their own fast local memory, and several processing elements. Each processing element has its own private memory, and are where actual computation is performed.

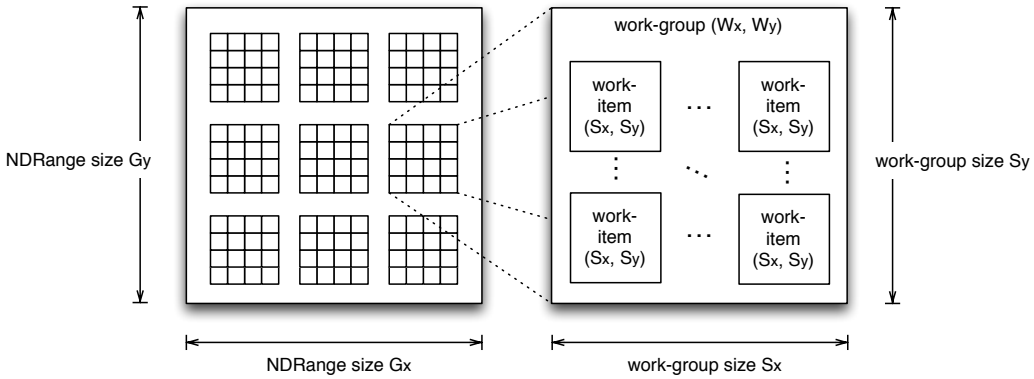


Figure 2.3: The OpenCL execution model showing work items and groups distributed over a two dimensional range. Each work item is given two identifiers, one global identifier specified by G_x and G_y, as well as a local identifier defined by S_x and S_y. Groups are also given identifiers in a similar manner.

The scheme OpenCL uses for the input range is called NDRange, which defines a N-dimensional range of integers where N is 1, 2 or 3. This input range is specified by a size and an offset, though this offset is typically zero.

2.3.2.1 Context and Command Queues

The host device is responsible for defining the context within which kernels will execute. Contexts include the following resources:

- **Devices**: The compute devices available for executing OpenCL kernels—these can be CPUs, GPUs and other devices,
- **Kernels**: The OpenCL functions which will be executed on a device
- **Program Objects**: The program source code and compiled executable that defines the kernels

- **Memory Objects**: A set of memory objects which are visible to the host and the devices. These memory objects usually contain the values and data on which kernels operate.

The host creates a context and manages these resources using calls in the OpenCL API. One of the structures created by the host is a command-queue which is used to synchronise operations. Commands are placed onto this command-queue and these are then dispatched to devices. Commands are executed asynchronously between the host and the device, though a programmer can explicitly synchronise to define the order, or specify a dependency tree of commands using events. Commands which can be dispatched onto the command-queue include:

- **Kernel execution commands**: Dispatch a kernel to execute on a device.
- **Memory commands**: Transfer data to and from memory objects, or additionally map and unmap memory objects.
- **Synchronisation commands**: To define the order of command execution.

2.3.3 The Memory Model

As shown in Figure 2.2 there are four regions of memory defined in the OpenCL platform model excluding host main memory. These regions are described as:

- **Global Memory**: This region of memory can be read and written from all work-items, this may be cached depending on the device.
- **Constant Memory**: A region of global memory that remains constant during the execution of a kernel. The host is responsible for initialising constant memory.
- **Local Memory**: This region of memory is local to a work-group. All work-items in the work-group may read and write from this memory. Depending on the device, this may be dedicated memory, or a region of global memory.
- **Private Memory**: This memory is private to a work-item, other work-items cannot access this.

2.3.4 A Kernel

Within the context of these models the actual specification of an OpenCL kernel is a C function notated with certain attributes specific to OpenCL such as `__kernel`, `__global` or `__local`. These functions are then stored on disk in textual form, and when they need to be used they are compiled by the OpenCL driver and then executed on the GPU.

A simple example is provided in Figures 2.4 and 2.5. Figure 2.4 shows a program which multiplies the elements of array A by 10, and then stores the result in array B. Figure 2.5 shows an implementation of an OpenCL kernel which will perform the multiplication on the GPU, essentially just the body of the loop. This kernel would be dispatched over the NDRange $(0..\|A\|)$ creating $\|A\|$ work items, each with an id which they can access through `get_global_id()` to index the data passed in as the parameters A and B.

```
1 public void times_10([int] A, [int] B):
2     for i in 0..|A|:
3         B[i] = A[i] * 10
```

Figure 2.4: A simple Whiley program which multiplies each element in array A by 10 and then stores the result in array B. This example also illustrates the use of pre-conditions in Whiley.

```
1 __kernel void times_10(__global int *A, __global int *B) {
2     int i = get_global_id(0);
3     B[i] = A[i] * 10;
4 }
```

Figure 2.5: A simple OpenCL kernel which multiplies each element in array A by 10 and then stores the result in array B. This is equivalent to the loop body in Figure 2.4.

2.4 Whiley

Whiley is a programming language developed by David Pearce which proposes to resolve issues with regard to the verification of object-oriented languages. Whiley addresses this by utilising an automated theorem prover to perform extended static checking and raise errors at compile time [36, 37].

As Whiley uses a verifying compiler, programs must be written with assertions that the compiler can verify to ensure that programs are correct. An example of these assertions is as follows:

```
1 int f(int x, int y) requires y != 0:
2     return (x*x) / y
```

The pre-condition in the previous code which asserts that y cannot be zero is necessary for the compiler to accept this program as correct. Without this statement the compiler would raise an error and point out that the code on line 2 could cause a divide by zero error.

To address the problems that many previous verifying compilers have faced Whiley has been made with several key design choices [36], two of which are of particular relevance to this project. These are the use of unbounded arithmetic and **call-by-value** semantics (§2.4.1).

The use of unbounded arithmetic avoids the issues faced in verifying programs which use modular arithmetic, where primitive types such as `int` have a fixed size and will wrap back to their minimum value if their maximum value is exceeded. In Whiley unbounded arithmetic applies to integers and reals which are implemented as unbounded rationals (fractions). Both these types are implemented in software so are much slower than their fixed precision hardware supported counterparts, and are only constrained in value by the amount of memory available to the program.

Also relevant to this project is the fact that Whiley is flow-typed, meaning that variables are declared without a type signature like in Java, OpenCL, C or many other languages. Instead the type of a variable is deduced from assignment statements, and at any point a variable will have the type of the value that was last assigned to it. For example:

```
1 void f(int x):
2     y = x
3     z = [1,2,3.14159]
4     z = 1.2
```

In this code the variable y has the type `int` as this is the type of x when it is assigned on line 2. However, `z` has two different types at different places in the program. Initially, on line 3 `z` has type `[int|real]`, however on line 4 this becomes `real` because of the assignment of the value `1.2` to this variable.

There are two types of loops available in Whiley, `while` loops and `for` loops. The syntax for both of these is as follows:

```
1  int sum_to_n_for(int n):
2      total = 0
3      for i in 1..n:
4          total = total + i
5      return total
```

```
1  int sum_to_n_while(int n):
2      total = 0
3      i = 1
4      while i < n + 1:
5          total = total + i
6          i = i + 1
7      return total
```

In the case of `for` loops the loop actually iterates over a list of numbers, rather than the traditional `for` loop which operates more like the `while` example and iterates a counter. The `1..n` syntax shown in the first example actually expresses the creation of a list containing these numbers, rather than a more efficient construct as in C or Java.

2.4.1 Relevant OpenCL Constraints

While OpenCL is based on C there are several constraints imposed by the language which pose problems for traditional approaches to implementing programs.

The first of these is that dynamic memory allocation is prohibited. This means that while a kernel is running, it cannot request more memory, all memory it uses must be preallocated by the host. This makes modifying data structures while the program is running difficult, as for example the process of appending an element to a list will require the list to expand to make room.

This constraint of not having dynamic memory allocation affects more than just modifying data structures, it also means that new data structures cannot be created, and existing data structures cannot be copied. This poses difficulties when implementing **call-by-value** function semantics in Whiley, as OpenCL uses **call-by-reference** like C. The difference between these two is seen when passing arguments to a function, with call-by-reference semantics any modifications performed to the data by the **called** function will affect the values passed in, meaning that the **caller** will have their data structures modified as well. With call-by-value this does not occur, the called function receives a copy of the arguments, meaning that the only way in which the called function can affect the caller is by returning the affected data structures as a return value.

The second constraint which poses a difficulty to implementing Whiley on a GPU is that when executing on the GPU it is not possible to perform any Input/Output (IO) such as printing messages for the user of the program to see, or to call any other functions which are not also running on the GPU—such as functions provided by the Java Whiley runtime.

OpenCL does also not support exceptions, or any form of early return which would allow all computation on the GPU to be halted by one of the threads. Examples of when this

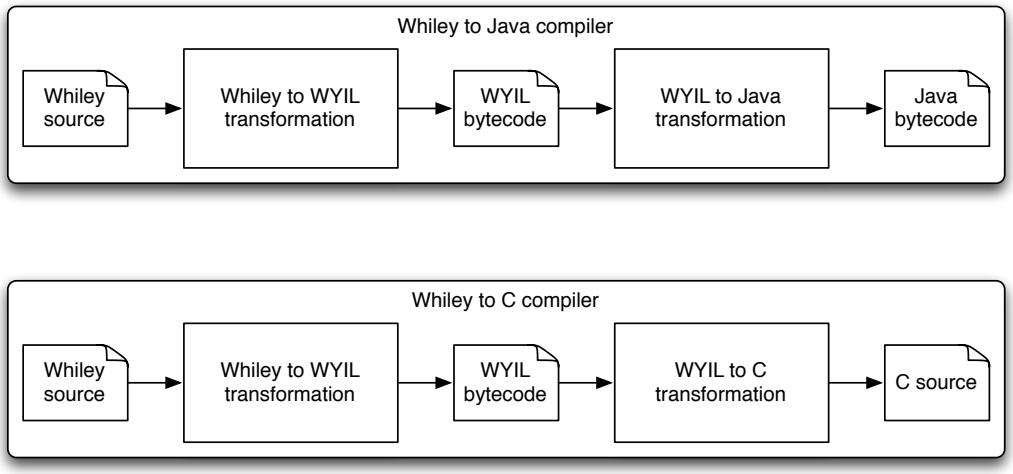


Figure 2.6: The build phases of the Whiley to Java and Whiley to C compilers.

would be desirable would be if a thread performed an illegal operation such as indexing out of the bounds of an array, or when dividing by zero.

The final constraint relevant to this project is the fact that arguments passed to a kernel must occupy flat regions of memory, which means that they cannot contain references to other regions of memory such as you would typically have when implementing data structures such as maps or lists of lists for example.

2.4.2 Compiler Architecture

The existing Whiley compiler utilises a chain of build phases to convert Whiley source code through various transformations into an output of Java bytecode or C source. The current build phases in the Whiley compiler are from Whiley source to Whiley Intermediate Language (WYIL), and WYIL to Java bytecode for the Whiley to Java compiler, and from Whiley source to WYIL, and WYIL to C source for the Whiley to C compiler as shown in Figure 2.6.

WYIL is a register based bytecode language developed specifically to support Whiley. WYIL has the same type system as Whiley, however, the language itself is much simpler and does not retain any of the higher level constructs such as `if-else` or `switch` statements—with the exception of loops. For example the Whiley program in Figure 2.4 becomes the WYIL shown in Figure 2.7.

Registers in WYIL are essentially numbered variables, where every Whiley variable and temporary value gets stored in a register. For example the expression `A[i] * 10` in Figure 2.7 does not use any variables, but in WYIL this becomes many lines with lots of temporary registers (Figure 2.7 lines 11 and 15–17).

2.5 Prior Work

Using the largely under-utilised resources of GPUs to accelerate the execution of programming languages is not a new idea. A large volume of research—especially in recent years—has been undertaken into different techniques for doing this, and the most effective ways of using GPUs for general computation.

Two different ways of approaching this problem have become evident in publications focusing on making GPU programming accessible to programmers. These are approaches

```
public void times_10(%0 [int], %1 [int]):
requires:
    lengthof %3 = %0 : [int]
    lengthof %5 = %1 : [int]
    asserteq %3, %5 "precondition not satisfied" : int
body:
    const %6 = 0 : int
    lengthof %8 = %0 : [int]
    range %9 = %6, %8 : [int]
    forall %10 in %9 (%1) : [int]
        const %19 = 0 : int
        assertge %10, %19 "index out of bounds (negative)" : int
        lengthof %20 = %0 : [int]
        assertlt %10, %20 "index out of bounds (not less than length)" : int
        indexof %14 = %0, %10 : [int]
        const %15 = 10 : int
        mul %16 = %14, %15 : int
        update %1[%10] %16 : [int] -> [int]
        nop
    return
```

Figure 2.7: The WYIL generated from compiling the Whiley code in Figure 2.4.

that attempt to transparently convert portions of code written in existing languages, and those that build a new framework or language tailored specifically to GPU acceleration. The former is the approach taken by this project, which means programmers need not concern themselves with the details of using a new framework. The latter approach is significantly easier to implement, but requires additional work from the programmer.

2.5.1 Transparently Converting Existing Languages to run on a GPU

Artigas et al. [4] describes a method for safe and alias-free regions of code which can be used to optimise and parallelise loops in Java. This technique uses runtime checks to determine whether the optimised or unoptimised code should be run depending on the possibilities for aliasing or exceptions.

Aliasing and exceptions are two features in many programming languages that cause problems when attempting to automatically parallelise code. Using Figure 2.4 we could trivially identify that the loop can be parallelised. However, if `A` and `B` can refer to the same or overlapping regions of memory (are aliased), then the behaviour of this program may be changed when run non-sequentially. Additionally if any operations inside the loop may throw an exception—such as indexing beyond the end of an array—then by executing in parallel the point at which the exception is thrown becomes undefined, which breaks exception semantics in most languages.

The optimisations to the compiler performed by Artigas et al. caused Java versions of benchmarks to perform at or above 80% the speed of equivalent highly-optimised Fortran versions.

This work was built on by Leung et al. [28], who develops automatic Just In Time (JIT) OpenCL acceleration of loops for a Java Virtual Machine (JVM). In this paper Leung et al. developed a classification system for different types of loops, as well as a cost model to

determine whether it is profitable to run a loop on the GPU. The four different types of loops identified by Leung et al. are:

1. **GPU-implicit loops**: A GPU-implicit loop executes the contents of the loop (the loop body) multiple times on the GPU, but the control flow of the loop does not appear explicitly anywhere in kernel. These are loops which can be converted directly from a loop to a kernel, where each work item corresponds to an execution of the loop body.

2. **GPU-explicit loops**: GPU-explicit loops contain the loop control flow explicitly in the code of the kernel. These are loops which are nested inside a GPU-implicit loop.

3. **CPU loops**: CPU loops are all loops which are not executed on the GPU. Any loop could be executed as a CPU loop, though this may be inefficient.

4. **Multi-pass loops**: Multi-pass loops are CPU loops which contain one or more GPU-implicit loops, which do not have the data they use modified by the CPU loop. Multi-pass loops are used by Leung et al. to optimise data-transfers. If a GPU-implicit loop will be executed multiple times and the results of its computations are not used by the CPU loop, there is no need to copy data from the GPU to the CPU and then back again for each iteration.

Leung et al. provides a general algorithm for determining loop types, which has been adapted to be used in this project. Empirical results have shown that the speedup over execution on the CPU ranges from 27% to 1,300%.

The cost model proposed by Leung et al. takes into account the relative speed of the GPU and CPU, and the size of the data to be transferred. The coefficients to this model are determined using least-squares regression and micro-benchmarks at JVM install time. Experiments have shown that this cost model is close to ideal, producing total execution times within 4.7% to 11.5% of the ideal time.

Chan and Abdelrahman [10] have developed a method of specifying what memory locations methods access, allowing them to build a system which uses this information to construct a data dependancy tree. In this system every method call runs on a new thread, and all these threads are managed by this dependancy tree. This approach has allowed their system to gain near-ideal performance on a four-processor Sun machine for a number of benchmarks.

2.5.2 Providing the Programmer with Additional Tools

Both Beck et al. [5] and Nystrom et al. [33] have focused on developing frameworks to allow programmers to more easily use GPU resources. Both of these projects have been developed for Scala and have taken the approach of using map and reduce operations [11] as the building blocks for parallel computation.

Beck et al. and Nystrom et al. have taken different approaches as to where to implement their language additions. Beck et al. have developed a compiler plugin which modified the Abstract Syntax Tree (AST) of the program as it is being compiled, whereas Nystrom et al. provide a framework which at run-time gets JVM bytecode for the map or reduce function being performed and then compiles a GPU version from this.

Beck et al. do not appear to have successfully completed the modifications to the Scala plugin, citing difficulties with updating the AST, and no noticeable speed improvements by executing on a GPU. This is in contrast to Nystrom et al. who have successfully implemented their modifications and have found code produced by their technique comparable in speed to equivalent native C code.

Chapter 3

Design

This project implements a solution to the problem of speeding up Whiley programs by extending the Whiley compiler to generate code targeting the OpenCL API. As OpenCL is the target language, devices besides GPUs can be used for execution though GPUs are the focus of this project.

In Whiley programs `for` loops are good candidates for parallelisation to take advantage of the power of GPUs. Each iteration of a loop executes the same instructions (the loop body), while operating on different data (the loop index and other information) similar to the SIMD architecture of GPUs. Unfortunately due to dependancies between iterations of the loop, or calls to functions not available on the GPU such as `println`, not all loops can be simply converted to run on a graphics card.

To address the correctness requirement the Whiley to OpenCL compiler has to analyse the Whiley program and select loops which could be successfully run on the GPU without changing their behaviour. This results in a program which partially runs on the CPU, and partially runs on the GPU as shown in Figure 3.1.

If a loop is analysed to be able to be parallelised, then the loop body is put inside an OpenCL kernel so it can be executed on the GPU. The remaining Whiley code must then be modified so loops which are executed on the GPU are replaced with calls to the OpenCL kernels.

3.1 System Architecture

The system architecture to implement this design involves three major architectural components, these are:

- a filter which edits Whiley code,
- a Whiley to OpenCL transformation module,
- a runtime environment which is responsible for managing the execution of kernels on the GPU.

These three components are located in two different parts of the project, the filter and the Whiley to OpenCL transformation are located inside the compiler, and the runtime support is embedded inside the compiled program.

The compiler itself follows the same architecture as the Whiley to C and Whiley JVM compilers (Figure 2.6), which allows it to reuse large portions of the existing code base. The new filter which has been added takes the output of the Whiley to WYIL build phase, and

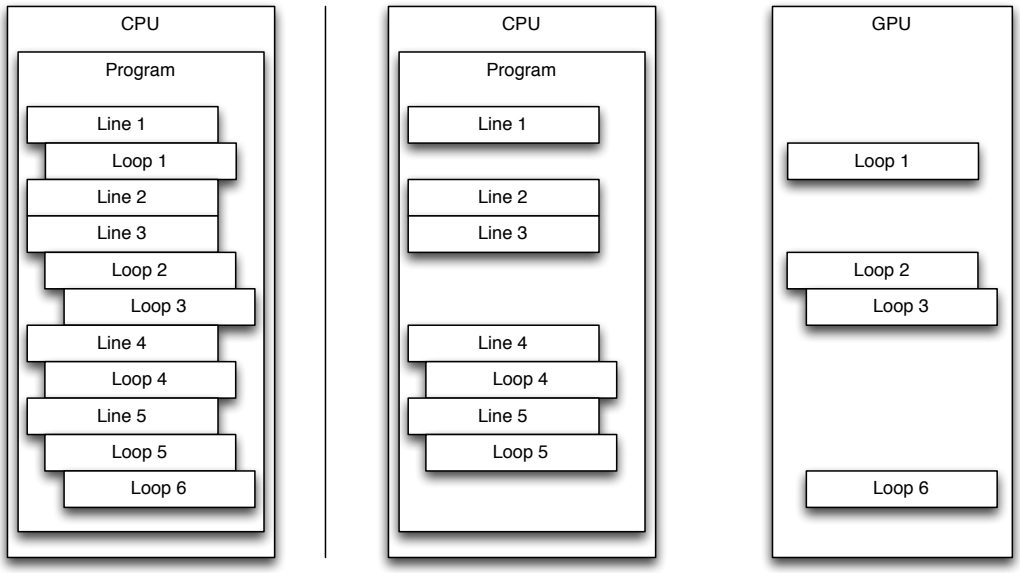


Figure 3.1: Comparison between how programs run using the vanilla Whiley compiler (left) to those that are compiled using this project (right).

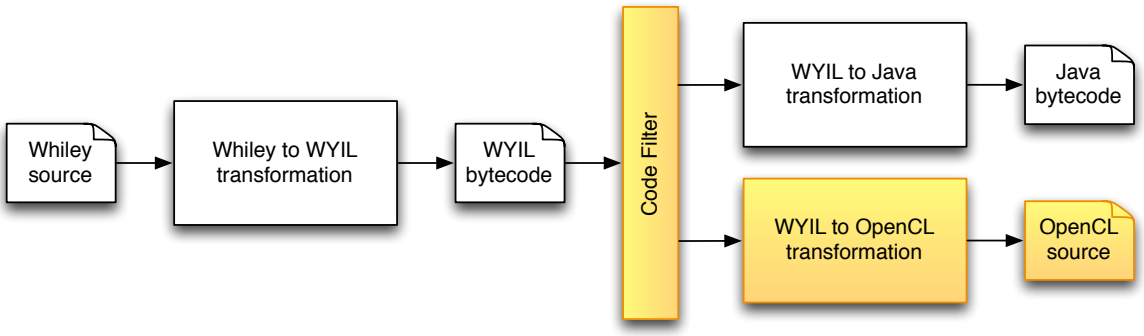


Figure 3.2: The build phases of the Whiley to OpenCL compiler. **Note** the highlighted components which have been added to the existing architecture (Figure 2.6).

determines which loops can be successfully modified to execute on a graphics card. The code is then split up into code which will run on the CPU, and the loops which will run on the GPU. The selected loops are converted into OpenCL and the remaining code which will run on the CPU is passed on to the existing parts of the compiler as shown in Figure 3.2.

3.2 Datatype Architecture

Due to the difference between Whiley types and those supported by OpenCL, there is no straightforward way to convert between the two. To ensure the two systems are interoperable a mapping has been created which defines how to perform the conversion. This mapping between Whiley types and OpenCL types can be separated into two parts, the mapping for primitive types such as integers, booleans and reals, and the mapping for composite types such as lists and tuples.

3.2.1 Primitive Types

For booleans the mapping from Whiley types to OpenCL types is straightforward, however for others—integers and reals—there is no direct mapping.

Booleans in Whiley can have one of two values, `true` or `false`. These two states can be simply encoded as `1` or `0` and be stored in an integer datatype such as an `unsigned char`. This is the approach taken by this project, and is a very common way of representing booleans in C-like languages [20, 2, 19].

Whiley integers cannot be fully represented in OpenCL because of constraints the language imposes (discussed in §2.4.1). Integers in Whiley are unbounded and can represent any integer value which means that as their value increases more memory is needed to store them. This expansion of memory requires dynamic memory allocation, which is not possible in OpenCL. This means that integers have to be stored in a fixed amount of memory, and if integers outside this range are used by the program behaviour will be undefined. 32bit `int`s are the largest integer type available on most GPUs, so these have been selected as the replacement for Whiley integers.

Reals in Whiley are also unable to be represented fully in OpenCL due to the lack of dynamic memory allocation. As reals are represented as fractions of Whiley integers, the numerator and denominator can become arbitrarily large in order to accurately represent a number. As this cannot be implemented on a GPU, again a fixed sized type must be used. The `float` type has been selected as the best replacement for reals, as it is the fastest and most precise type available on most GPUs which is capable of representing decimal numbers. `float`s store decimal numbers in a different format to that of Whiley reals, and cannot represent every possible Whiley real number. Thus converting from a real to a `float` and back may result in a slightly different value. Using double-precision numbers would improve this somewhat, however, the `double` type is not supported on all graphics hardware, and is optional in the OpenCL specification [25].

Despite the downsides of these type mappings, using native types supported by the GPU has the benefit of substantially improving performance. The difference between the datatypes used in Whiley and those used on the GPU does change the semantics of Whiley programs. However, in many cases when programs are run on the GPU performance is of higher concern than precision. A summary of the different types and how these are mapped between Whiley and OpenCL can be seen in Table 3.1.

3.2.2 Composite Types

Whiley supports composite types such as lists, tuples and more, all of which require a different storage format when located on the GPU. Because dynamic memory allocation is not available on the GPU, composite types must also be restricted to a fixed size. Additionally OpenCL kernels can only be given flat regions of memory, so all data structures accessed must be stored in one contiguous block. This complicates the usage of all composite datatypes as Whiley does not represent them in this manner.

For these reasons only three composite datatypes are supported by this project, lists, lists of lists and tuples, all of which must have primitive element types.

3.2.2.1 Lists

Lists in Whiley are stored as a length, and a sequence of references to elements which can be directly indexed using an integer. This object structure cannot be uploaded to the GPU, so a different flat format is used. To be able to store this information the size of each element must be known, and must be the same for all elements, otherwise the fixed size of the list

Type	Whiley on the JVM	OpenCL	Comments
Integer	`BigInteger`	`int`	`int`s in OpenCL are 32bit signed integers, Only values from -2,147,483,648 to +2,147,483,647 can be represented. Whiley integers larger than this, or arithmetic operations with results larger than this will produce incorrect results on the GPU.
Real	`WyRat`	`float`	`WyRat` is a custom software implementation of arbitrary precision rational numbers. As these are not supported on the GPU `float`s are used instead. In OpenCL `float`s are IEEE 754 single-precision numbers, which means that a large amount of precision may be lost during this conversion.
Boolean	`Boolean`	`unsigned char`	Booleans behave exactly the same in the OpenCL implementation as in Whiley.

Table 3.1: The mapping of primitive types from Whiley to OpenCL. **Note** that only booleans have all of their behaviour preserved.

cannot be computed in advance of upload to the GPU. The format for the flattened list is that of a traditional array in C-like languages, a contiguous block of memory the size of each element times the number of elements, with the exception of the first element which is an integer, and is used to store the size of the list. An example of the difference between the Whiley list and the GPU format can be seen in Figure 3.3.

3.2.2.2 Lists of Lists

The one exception to the rule that lists must have only primitive element types is for lists which contains lists. These lists of lists—multidimensional arrays—are specially treated to ensure they can be used on the GPU. This datatype is supported by the Whiley to OpenCL compiler because it is a common structure in many programs which parallelise well, and supporting these arrays allows optimisations to be applied by the compiler which can improve the performance of code running on the GPU.

Multidimensional arrays must be resized so that each axis is the same length–this can be thought of as making the multidimensional array rectangular—which is necessary to ensure correct element indexing. The array is then laid out flat into one-dimensional form innermost elements first as can be seen in Figure 3.4.

Because of the ordering used by the flattening operation, an access into a multidimensional array can be easily mapped to an equivalent index of the one-dimensional array. For example the index `array[y][x]` is mapped to `array[y * width + x]`. This easily generalises to higher dimensions of multidimensional arrays, such as lists of lists of lists and this is made use of within the compiler.

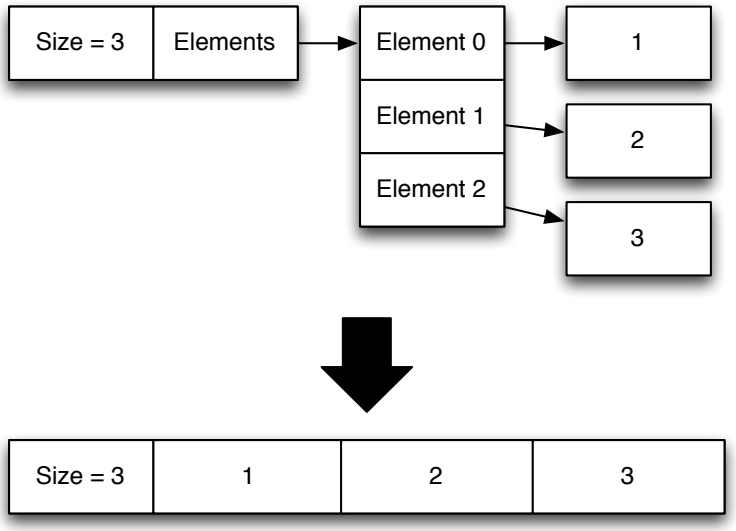


Figure 3.3: An example of the difference between data structures in Whiley (top) and those in OpenCL (bottom). **Note** how the Whiley structure involves many connected regions of memory, while the OpenCL version occupies only one.

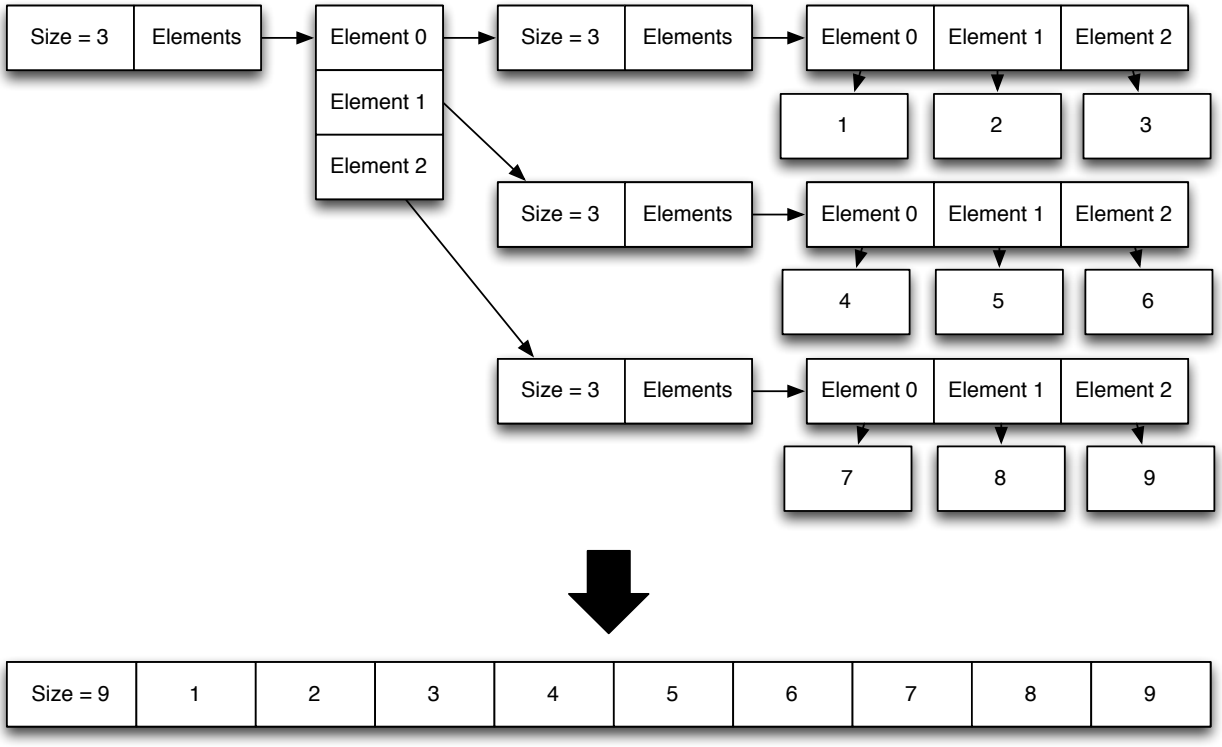


Figure 3.4: The multidimensional array [[1,2,3],[4,5,6],[7,8,9]] in Whiley (top) and in OpenCL (bottom). **Note** how the information about the size of each dimension is lost, this must be separately passed into the OpenCL kernel.

3.2.2.3 Tuples

In Whiley tuples are accessed and updated internally using an index, just as if they were lists. For this reason the OpenCL in-memory representation of this type is identical to that of lists, simplifying the logic needed in the OpenCL kernel to access these data structures. While this is desirable, it does impose the same restrictions as those for lists, namely that the type of each element in the tuple must be the same.

Chapter 4

Basic Implementation

The three major architectural components of the OpenCL/GPU support in Whiley added by this project are covered in the following sections. §4.1 describes the filter which edits Whiley code, §4.2 details the inner workings of the OpenCL transformation module, and §4.3 explains what the runtime support does.

4.1 Code filter

The code filter is the largest and most complex component of the Whiley to OpenCL compiler. This component is responsible for the following:

1. selecting which loops to convert for execution on the GPU,
2. applying optimisations to improve the performance characteristics of these loops when executing on GPUs,
3. modifying these loops so they can be executed on a GPU,
4. injecting the relevant code to replace the loops such that at runtime the appropriate OpenCL kernels are executed on the GPU.

The first two of these points are very involved, and are covered separately in Chapters 5 and 6, while the third point is covered below. An overview of the way these parts fit into the architecture of the filter can be seen in Figure 4.1, and an example of how it operates is covered in §4.1.2.

4.1.1 Loop Modifications

Generally Whiley loops cannot be simply converted to run on a GPU, instead several modifications are required. The three modifications which the code filter performs are as follows:

- removing any non-critical code which cannot be implemented in OpenCL, such as assertions,
- removing any code which is never used,
- flattening multidimensional arrays to one dimensional arrays, and updating any code which accesses these arrays so it still functions correctly.

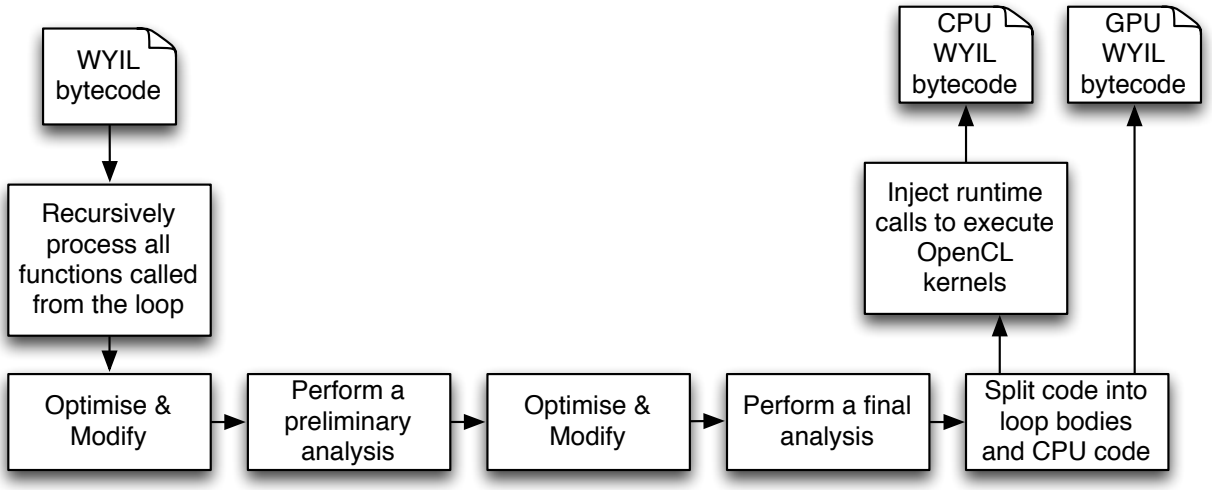


Figure 4.1: The dataflow architecture of the code filter.

Removing non-critical code and code which is never used are two simple modifications which produce only a small impact on the behaviour of the loop, but significantly improve the number of loops which are able to be executed on the GPU. For example assertions are commonly inserted by the Whiley compiler to ensure that array accesses do not go out of bounds or divisions by zero are not performed, however if the assertion condition is not met an exception is raised which renders the code incompatible with OpenCL. In this example removing the assertion means that indexing outside the bounds of an array would result in undefined behaviour, and while this is not desirable, this is a sacrifice which must be made to be able to execute the majority of compatible Whiley programs on a GPU.

4.1.1.1 Flattening Multidimensional Arrays

As described in the section on mapping between Whiley types and OpenCL types (§3.2), multidimensional arrays have to be flattened and converted into one-dimensional arrays before they can be uploaded to the GPU. The code filter is the only part of the compiler which is capable of doing this, as the code which accesses these arrays also needs to be modified.

The code filter accomplishes multidimensional array flattening by performing the following four actions:

- Identifying all the multidimensional arrays that are accessed from inside the selected loops.
- Computing the dimensionality of each of the multidimensional arrays.
- Injecting calls to the runtime before and after the loop. These calls will perform the actual array flattening operation using the computed array dimensionality.
- Modifying all array accesses that happen between these injected calls such that they remain correct with the altered data layout.

```
public void times_10(%0 [int],
                     %1 [int]):
body:
    const %6 = 0 : int
    lengthof %8 = %0 : [int]
    range %9 = %6, %8 : [int]
    (* insert code to
        call kernel over the
        array %9 and with the
        arguments %0 and %1 *)
    (* insert code to
        get results of
        kernel *)
    return
```

(a) For the CPU

```
forall %10 in %9 (%1) : [int]
    indexof %14 = %0, %10 : [int]
    const %15 = 10 : int
    mul %16 = %14, %15 : int
    update %1[%10] %16 : [int] -> [int]
```

(b) For the GPU

Figure 4.2: The WYIL generated from compiling the Whiley code in Figure 2.4 after filtering. The code before filtering is shown in Figure 2.7. **Note** how the GPU code is the loop from Figure 2.4 and the CPU code is the remainder of the program—with some minor changes.

4.1.2 Code Filter Example

To illustrate what changes this code filter actually performs we will take a detailed look at what it does to the Whiley program shown in Figure 2.4. For this example optimisations will be ignored as they are separately covered in Chapter 6.

The first stage of processing is for the Whiley program to be converted into the simplified WYIL language (see Figure 3.2). Once this is done the code filter will be passed the WYIL code—shown in Figure 2.7—and begin editing it.

The code filter then performs the processing discussed above and shown in Figure 4.1. The results of this processing are shown in Figure 4.2, and the steps involved in producing this are detailed below:

1. The code is optimised and modified to improve the number of loops able to be executed on the GPU. This results in all the assertions in Figure 2.7 being removed.
2. The code is analysed to determine which loops are compatible with running on the GPU (Chapter 5). This analysis determines that the loop can be run on the GPU.
3. The code is optimised and modified a second time. This results in no changes as optimisations are ignored in this example.
4. A final analysis to select loops is performed. No changes are made to the selection.
5. The code is split into CPU code (Figure 4.2a) and the loops to execute on the GPU (Figure 4.2b).
6. The CPU code is edited to replace the selected loops with calls to the runtime shown as comments in Figure 4.2b.

After this has been done the filtered code for the CPU is passed to the existing WYIL to JVM part of the pipeline, and the code for the GPU is passed to the WYIL to OpenCL transformation module which is discussed below.

```
1  __kernel void whiley_gpgpu_func_0(__global int *r0_16Yk41, __global int *r1_16Yk41) {
2      // Beginning Boilerplate
3      // Begin register 0 list unpacking
4      int r0_16Yk41_size = r0_16Yk41?((__global int *)r0_16Yk41)[0]:0;
5      r0_16Yk41 = (__global int *)(((__global int *)r0_16Yk41) + 1);
6      // End register 0 list unpacking
7      // Begin register 1 list unpacking
8      int r1_16Yk41_size = r1_16Yk41?((__global int *)r1_16Yk41)[0]:0;
9      r1_16Yk41 = (__global int *)(((__global int *)r1_16Yk41) + 1);
10     // End register 1 list unpacking
11     int r10_l4D;
12     int r14_l4D;
13     int r15_l4D;
14     int r16_l4D;
15     // Ending Boilerplate
16
17     // Begin kernel
18     (r10_l4D) = get_global_id(0); // Get work item
19     (r14_l4D) = (r0_16Yk41[(r10_l4D)]); // indexof %14 = %0, %10 : [int]
20     (r15_l4D) = 10; // const %15 = 10 : int
21     (r16_l4D) = ((r14_l4D) * (r15_l4D)); // mul %16 = %14, %15 : int
22     (r1_16Yk41[(r10_l4D)]) = (r16_l4D); // update %1[%10] %16 : [int] -> [int]
23     return;
24 }
```

Figure 4.3: Simplified generated output from the Whiley code seen in Figure 2.4. The bytecodes from Figure 4.2b are shown in the comments of the OpenCL code on lines 18–22.

4.2 WYIL to OpenCL Transformation

Even though OpenCL is a C based programming language, the existing C backend could not be used for the WYIL to OpenCL transformation as it relies on many language features which do not exist in OpenCL such as dynamic memory allocation and system library calls. Instead an entirely different implementation has been developed that maps each WYIL bytecode from a selected loop's body to one or more lines of OpenCL code in a linear fashion. For example it can be seen that lines 19–22 of the OpenCL kernel shown in Figure 4.3 are the direct equivalent of the loop body in Figure 4.2b.

The output from this conversion process is then injected into an OpenCL kernel which is identified by the module name of the source file, and a unique kernelID. Every variable accessed from the loop body but defined outside the loop is turned into a kernel parameter which will be passed in when the kernel is executed—again this can be seen in line 1, Figure 4.3.

4.2.1 Transcription

By the time the WYIL to OpenCL transformation module gets passed the code which will be executed on the GPU, all the incompatible bytecodes have been removed by the filter (§4.1). Each WYIL bytecode performs one action, e.g. adding two registers, accessing a list, jumping to a line, all of which can be implemented easily in OpenCL. As an example a multiply bytecode like so:

```
1 mul %16 = %14, %15 (* Multiply register 14 with register 15 and
2                       store the result in register 16 *)
```

becomes the following OpenCL:

```
1 register16 = register14 * resister15;
```

However, Figure 4.3 shows on line 21 that the actual generated code is not nearly as tidy, in fact it becomes:

```
1 (r16_l4D) = ((r14_l4D) * (r15_l4D));
```

The reason for this is twofold. Firstly extra parentheses are added by the automatic code generator to be certain that the evaluation order is as intended (this is not needed for this example but is in other cases) and secondly that WYIL has two language features which are not natively supported by OpenCL. These WYIL language features mean that registers and functions need to be renamed so that they work correctly in OpenCL.

4.2.1.1 Register Renaming

Each register in WYIL is equivalent to a Whiley variable, and these registers maintain the property of Whiley variables in that they are **flow-typed**. This means that during the execution of a section of code the type of a register can change. This poses a problem as each WYIL register has to be mapped to a OpenCL variable, however OpenCL variables cannot change types. The approach this project takes is to analyse the program to determine which types each register will be used for, and then declare an OpenCL variable for each register and each type (lines 11–14 of Figure 4.3). Whenever a register is then accessed its type at that point is checked to ensure that the correct OpenCL variable is used. For example if register 5 has two types, `int` and `real`, then the following OpenCL variable declarations will be used:

```
1 int r5_l4D;
2 float r5_05Wz;
```

While this naming scheme is not particularly human readable, it uses the same method used throughout the entire Whiley compiler for representing types as strings (`l4D` is `int` and `05Wz` is `real`).

4.2.1.2 Function Renaming

Whiley allows different functions to share the same name provided that the parameters to the function are different (**function overloading**). For instance a program containing the following two functions is perfectly valid:

```
1 public int f(int x):
2 public int f(real x):
```

In OpenCL however this is not allowed, every function must have a distinct name. This means that a similar technique to the one for solving the register problem is used. This technique is called **name mangling** and involves adding the type of the function to the function name, for instance the first example above becomes the following:

```
1 int f_Y9bFXA$W(int x);
```

This again results in less readable code, however it ensures that every function will have a distinct name.

4.2.2 Readability

As can be seen in Figure 4.3 and previous examples the generated output from compiling a Whiley program is largely unreadable, and barely resembles the original Whiley code (Figure 2.4). This is because the OpenCL conversion takes in WYIL bytecodes (Figure 2.7) which do not have any information about many of the high-level language structures such as `if` statements and variables names. For example `if` statements are replaced with jumps in the Whiley to WYIL build phase, meaning that the OpenCL code contains `goto`s rather than the higher level constructs which could be implemented. Additional metadata or reverse-engineering would be required to produce structured and easily readable OpenCL code.

4.2.3 Loop Indexes

As an OpenCL kernel function is the body of a selected loop, the code contained within it needs access to the index variable. To get this the inbuilt OpenCL function `get_global_id()` is used. This function allows a work item to get its global ID or location within the range of executing kernels (§2.3.2).

When the OpenCL kernel is the result of selecting a standard `for` loop for execution on the GPU, this kernel index is used to load the appropriate element from the list the loop is iterating over and store it into the variable representing the loop index. A simplified example of this would be:

```
1  __kernel void my_kernel(__global int *my_source_array) {
2      int my_loop_index = my_source_array[get_global_id(0)];
3      ...
4  }
```

Kernels originating from loops which iterate over a range and have been optimised (§6.1.1) operate in a similar manner, except the kernel index is stored directly into the index variable. This simplifies the previous example to:

```
1  __kernel void my_kernel() {
2      int my_loop_index = get_global_id(0);
3      ...
4  }
```

In the case where the `for` loop was a multidimensional loop (§6.2.1), the index for each different dimension can be accessed by calling the `get_global_id()` function with the dimension number as the argument. Supporting multidimensional loops requires minimal changes to our example:

```
1  __kernel void my_kernel() {
2      int my_outer_loop_index = get_global_id(0);
3      int my_inner_loop_index = get_global_id(1);
4      ...
5  }
```

4.2.4 Function Calls

Any functions called from the body of the loop also need to be transcribed to OpenCL, though this has some associated constraints. Rather than the call-by-value semantics of Whiley, these functions in OpenCL have call-by-reference semantics, meaning that only

functions which do not make use of call-by-value semantics can be correctly run in OpenCL. Currently the compiler does not enforce this restriction, so the behaviour of functions which do rely on these semantics is undefined.

While ideally these semantics would be preserved by the OpenCL conversion, call-by-value semantics cannot be easily implemented on the GPU. To implement call-by-value semantics a copy of the arguments must be made before calling a function, which requires dynamic memory allocation which is not available on the GPU.

4.3 Whiley OpenCL Runtime Support

For a Whiley program to be able to communicate with the GPU it has to use the OpenCL drivers installed on the computer. Whiley has no inbuilt method for doing this, so a runtime library is provided to the program when running on the JVM which allows it to connect to the driver. This runtime support is implemented as a set of native functions in the Whiley runtime library and a dependency on the Java OpenCL Bindings (JOCL) library [22].

There are two different mechanisms provided by the runtime to execute OpenCL kernels, either by dispatching a kernel over an array, or over a range. These two methods are used in conjunction with the different ways in which loops are indexed (§4.2.3) and optimisations which may be applied (Chapter 6). Additionally ranges can be multidimensional, which is used in the case when multiple nested loops are flattened by the multidimensional loop flattening optimisation (§6.2.1).

These OpenCL runtime functions are provided with the name of the OpenCL file to load, and the unique name of the specific kernel which should be invoked. These two functions have to perform seven tasks:

1. the OpenCL kernel source is loaded from disk, compiled and linked,
2. the parameters to the kernel are **marshalled** (§4.3.1),
3. the marshalled data is uploaded to global memory on the GPU (§2.3.3),
4. the OpenCL kernel is dispatched to execute over the uploaded data,
5. the data is downloaded,
6. the data is **unmarshalled** (§4.3.1),
7. the unmarshalled data is returned from the function so that the changes made by the kernel are reflected on the host CPU.

As the first initialisation phase of compiling and linking the program can be very costly, compiled OpenCL programs are cached for later reuse. When the same file name and kernel name are used in a runtime call, the cached OpenCL kernel is immediately available.

4.3.1 Marshalling and Unmarshalling

The process of converting objects on the host into a form that can be uploaded to the GPU is termed **marshalling**, while conversion in the other direction is called **unmarshalling**. In this project these two processes involve converting between Java objects on the JVM heap, and the data format specified in §3.2 which can be interpreted by the GPU.

Chapter 5

Loop Selection

Determining which loops can be successfully converted to run on a GPU is crucial to this project, as without it certain loops would behave incorrectly. The loop selection mechanism has to check the following conditions to determine if a loop is compatible:

1. whether the loop contains any bytecodes which cannot be converted to OpenCL (for example assertions which raise exceptions),
2. whether there is any control flow which jumps out of the loop—such as `break`, `continue` or `return` statements—requiring an early return from the kernel (not supported by OpenCL),
3. whether there are any data dependancies between consecutive executions of the loop body,
4. whether the data structures accessed from the loop body are supported by the type mapping (§3.2),
5. whether all the functions called from the loop body are also compatible.

All of these conditions can be easily checked by simply inspecting the loop body, with the exception of the third condition. Determining whether an arbitrary loop behaves the same when it is parallelised is generally undecidable [34, 9, 8, 23, 39], so constraints have to be placed on the types of loops which are supported by this analysis.

5.1 Data Dependancy Analysis

When loops are incompatible with being parallelised because of data dependancies, this means that an iteration of the loop depends on a previous iteration. For example a loop which sums the numbers in an array needs to use the value calculated for the previous iteration and add the new value to it:

```
1 sum = 0
2 for x in my_numbers:
3     sum = sum + x
```

When run in parallel loops which have these dependancies behave incorrectly, as the values which a loop iteration depends on may not have been calculated yet. With the previous example if the loop was run in parallel all the iterations will read `sum` as being zero, add

x to it and then update `sum`. At this point `sum` will contain zero + one of the `x` values, not the sum of all the elements.

These problems (called **data-races** [3]) occur anywhere there are accesses to the same variable from different threads and at least one of the accesses is a write [40]. As each iteration of the loop will run in a different thread on the GPU, if an iteration writes to a variable or element in an array, no other iterations can access the same location or the loop will not execute correctly. To ensure that this does not happen the compiler needs to analyse all the loops and determine which ones do not have this problem.

To perform this analysis there are two operations that need to be executed, a data dependency analysis phase which determines for each loop whether it could be run in parallel, and then a loop categorisation stage which classifies each loop depending on where it could appear in the compiled output.

The data dependency analysis performed by the Whiley to OpenCL compiler is fairly straightforward. For every loop which may be executed on the GPU the set of all registers and array elements which are shared between loop iterations is collected. Only the locations which are written to from inside the loop could potentially cause race conditions, either when a read and write occurs in the wrong order, or if two writes conflict. Because of this if a loop contains any writes to shared locations, it is determined to not be compatible with execution on the GPU..

If an array is written to, further analysis is required. Determining whether arbitrary indexes into an array are actually independent is an extremely complex task, for instance to determine whether the following code accesses the same loop index twice requires difficult program analysis, even though a human can easily identify they are the same:

```
1  for i in 0..n:
2      my_array[(10 * i) + 10] = ...
3      my_array[10 * (i + 1)] = ...
```

To avoid this problem constraints have been placed on the ways in which arrays can be indexed. For a loop to be compatible all array indexes must be done by directly using the loop index, and the loop must iterate over a range of numbers rather than an arbitrary array of data. To illustrate, only loops of the following form are supported:

```
1  for i in 0..n:
2      my_array[i] = ...
```

Once an array access has been shown to be compatible, the exact index used to access the array is stored and all other indexes into the array for reading or writing must be the same.

If all of these conditions have been met for all the variables and arrays accessed inside the loop, then it will be safe to parallelise and run using OpenCL.

5.1.1 Categorising Loop Types

After loops have been analysed for compatibility they are classified into four different types in a manner similar to Leung et al. [28] as described in §2.5.1. The first three loop types—GPU-implicit, GPU-explicit and CPU—are the same, and the forth is a new category GPU-implicit-inner. These categories are illustrated in Figure 5.1 and are described as follows:

1. **GPU-implicit loops**: A GPU-implicit loop is a loop which will have its body converted to the contents of the OpenCL kernel. GPU-implicit loops are identified in the program as loops which are compatible with being executed in parallel, and which do not occur inside another GPU-implicit loop.

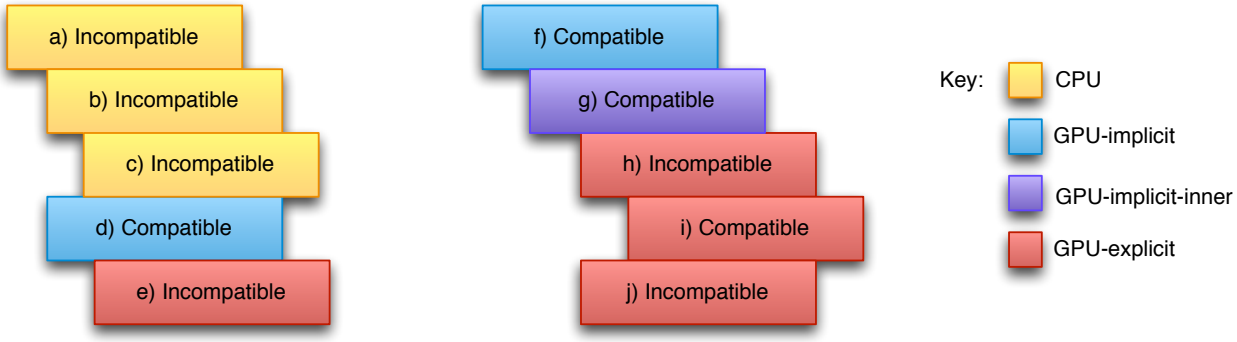


Figure 5.1: An illustration of how loop categories are determined based of whether they are compatible with being parallelised, and where they are located with respect to other loops. **Note** how despite (i) being compatible it does not become GPU-implicit-inner because it is contained within a GPU-explicit loop and itself becomes GPU-explicit.

2. **GPU-implicit-inner loops**: GPU-implicit-inner are loops which could participate in the multidimensional loop flattening optimisation, and become part of a multidimensional loop which is executed on the GPU. These loops are loops which are compatible with being parallelised and occur directly inside a GPU-implicit or another GPU-implicit-inner loop without depending on it.

3. **GPU-explicit loops**: GPU-explicit loops are loops which will explicitly occur in the OpenCL output. These are any loops which occur inside a GPU-implicit loop, but which do not meet the conditions for a GPU-implicit-inner loop.

4. **CPU loops**: CPU loops are all loops which are not executed on the GPU. These are any loops which are not parallelisable, and do not occur inside a GPU-implicit loop.

Loops are classified in this way so that later stages in the compiler are able to quickly determine what type each loop is, ensuring that modifications are only performed on loops which are suitable. This is especially relevant for optimisations, as these require detailed information about loops to be able to correctly operate.

Chapter 6

Optimisations

Because of the differences between the architectures of CPUs and GPUs, the same code will perform differently when run on these devices. Some code which will run fast on a CPU will perform very poorly on a GPU, and the same is true in reverse. Because Whiley is a programming language which is designed to run on a CPU developers will typically not write programs which have structures which suite GPUs. In fact programs which are written for GPUs have to be specifically tailored to get the best possible performance out of the graphics hardware.

To be able to get good performance out of the graphics hardware, the code taken from the Whiley program which will be run on the GPU has to be modified. Because GPUs are so sensitive to program behaviour, there are very few generic optimisations which can be easily applied and will likely result in better performance. The two optimisations which have been implemented in the compiler are intended to reduce the amount of data which needs to be transferred to/from the GPU, and to increase the amount of parallelism exploited by the program.

6.1 Data Transfer Reduction

Reducing the amount of data that has to be transferred to and from the GPU is a straightforward way in which the performance of the program can be improved. When running a program on the CPU it is very cheap to allocate and use memory due to the high speed connection between the CPU and main memory.

When running a kernel on the GPU memory needs to be allocated by the host and then uploaded to the GPU. As well as requiring a data transfer and an extra allocation, this also requires marshalling and unmarshalling due to the differences between Whiley and OpenCL (§4.3.1).

6.1.1 Forall to For Loop Optimisation

As mentioned in §2.4 `for` loops in Whiley are the equivalent of `forall` loops in other languages, meaning that they can only be used to iterate over data structures such as lists. For example to loop over the numbers between 0 and 100, a Whiley program would create a list containing all these numbers and then use a `for` loop to iterate over it. This behaviour is clearly sub-optimal in the case where the loop is iterating over a range as all the numbers in the list are known beforehand and can be very easily computed using a normal loop.

In the Whiley to OpenCL compiler normal Whiley `for` loops (hereafter referred to as `forall` loops) are handled by uploading the list that will be looped over to the GPU, and

```
1 array = 1..10
2 for x in array:
3     <loop body>
```

(a) Whiley.

```
1 for(int x = 1; x < 10; x++) {
2     <loop body>
3 }
```

(b) OpenCL.

Figure 6.1: A loop compatible with the `forall` to `for` loop optimisation (6.1a) and the resultant OpenCL code (6.1b). **Note** that this optimisation is the cause of the different loop indexing methods discussed in §4.2.3.

```
1 array = 1..10
2 array[0] = 22
3 for x in array:
4     <loop body>
```

(c) Whiley. **Note** that line 2 is what prevents the optimisation from being applied.

```
1 for(int x_index = 0;
2         x_index < array_size;
3         x_index++) {
4     x = array[x_index];
5     <loop body>
6 }
```

(d) OpenCL. `list` and `list_size` are provided to the kernel as arguments.

Figure 6.2: A loop not compatible with the `forall` to `for` loop optimisation (6.1c) and the resultant OpenCL code (6.1d).

then the OpenCL kernel indexes into this array to get the value of the element, which will in many cases be the same as the index. This requires the transfer of a large amount of redundant data to the GPU in the case where the loop is over a range.

To deal with this particular situation an optimisation stage in the Whiley to OpenCL compiler detects all instances of `forall` loops iterating over ranges, and replaces these with a standard OpenCL `for` loop. Examples of loops which can and cannot be optimised can be seen in Figure 6.1 and 6.2.

6.2 Number of Threads

GPUs are used to process millions of pixels in fractions of seconds, so they are designed to process millions of work items efficiently. If a GPU is only provided with a few thousand threads it may not be able to fully exploit the hardware to produce the desired performance. Providing the GPU and the OpenCL driver with a larger number of threads generally provides more opportunities for them to better use the available hardware resources.

This is the reason the multidimensional loop flattening optimisation is applied to selected loops, it allows the compiler to generate code which will produce a greater number of OpenCL work items.

6.2.1 Multidimensional Loop Flattening

Because this project focuses on executing loops on the GPU, the way in which the compiler increases the number of threads that are run on the GPU is by running multiple loops simultaneously. However, because the body of each thread must have the same OpenCL code, the only way this can be done is if the two loops are nested inside each other.

This optimisation allows the compiler to do this by flattening nested loops together, so that the body of the innermost loop is executed as many times as the inside loops executes,

```
for y in 0..1000:
    # <code 1>
    for x in 0..1000:
        # <code 2>
    # <code 3>
```

(a) Before.

```
for x in 0..1000 and y in 0..1000:
    # <code 1>
    # <code 2>
    # <code 3>
```

(b) After.

Figure 6.3: The modification that the multidimensional loop flattening optimisation performs. **Note** how the number of iterations per loop (and therefore threads) increases from 1,000 to 1,000,000.

times the number of times the outer loop executes.

For example if there is a piece of code which accesses all the pixels in an image using a loop which iterates over y, and a loop which iterates over x inside it, this would become a loop which iterates over both x and y at the same time (Figure 6.3a). As can be seen in Figure 6.3b this increases the number of threads from the size of x, to the size of x times y.

To be able to perform this optimisation, the compiler needs to be able to determine which loops can actually be merged together. This check can be done simply by using the loop types which were computed earlier (§5.1.1). The outer loop must be classified as a GPU-implicit loop, and the inner loop must be classified as GPU-implicit-inner, as this means that both loops are able to be parallelised. The only other check that must be performed is to check whether the inner loop depends on any results computed by the outer loop, as the structure of the loop is changed. The different ways in which two nested loops are processed depending on their loop classification is shown in Figure 6.4, this clearly illustrates that when the optimisation can be performed the improvement in the number of threads used is dramatic.

To actually perform the optimisation, the compiler takes the body of the inner loop, and replaces the inner loop with this code. Once this is done the range that the inner loop loops over is added to the outer loop, and it is converted into a multidimensional loop. This process can be seen in Figure 6.3.

This optimisation does introduce the issue that some portion of the code will be executed more times than the programmer intended. For example `<code 1>` of Figure 6.3 goes from being executed 1,000 times to being executed 1,000,000 times. However because of the loop categorisation it is known that this code is not depended on by the inner loop, and doesn't affect any shared variables. This means that the duplicated computation will not change the behaviour of the program. In most cases this duplicated computation does not negatively affect the execution time because the benefit of using more threads outweighs the cost of repeated computation.

Once this optimisation has been applied the OpenCL kernel which is produced by the compiler can be dispatched over each dimension simultaneously, drastically improving the number of threads which are executed on the GPU. This optimisation easily generalises to more dimensions, so in the cases where more than two compatible loops are nested the benefits are even greater.

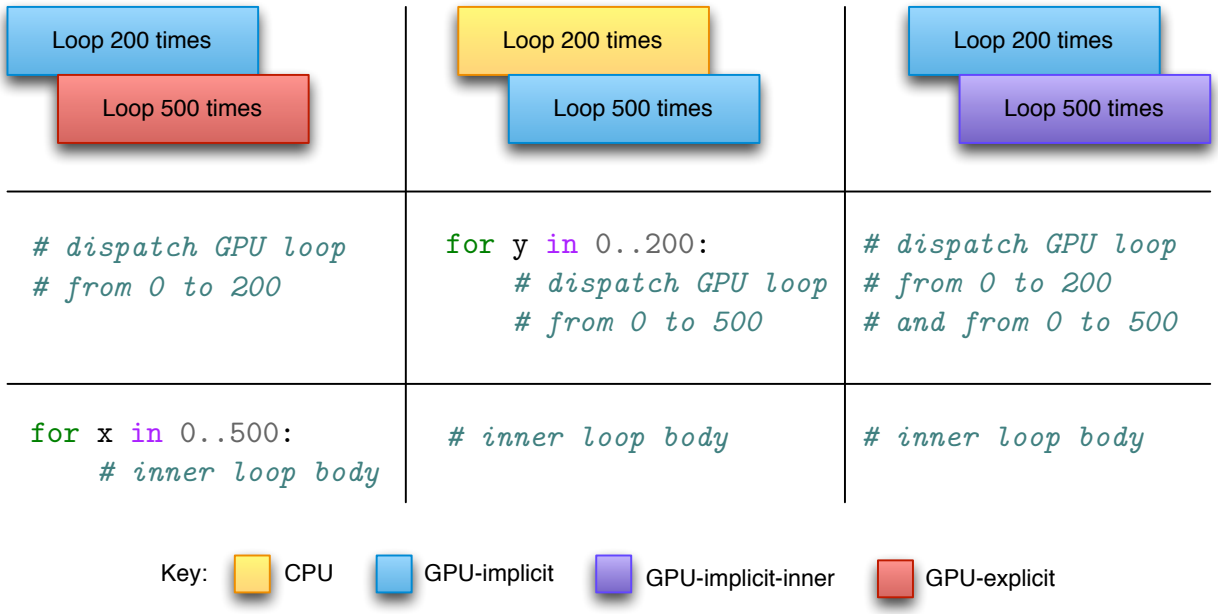


Figure 6.4: The different possible ways in which two nested loops would be executed depending on classification. **Top** row is loop type classification, **Middle** is code which executes on the CPU and **Bottom** is code which executes on the GPU. **Left (without optimisation)** the outer loop is executed on the GPU and the inner loop is repeated 200 times. **Centre** the outer loop is executed on the CPU and the body of the inner loop is dispatched to the GPU 500 times. **Right (with optimisation)** both loops are executed on the GPU and the body of the inner loop is dispatched 100,000 times.

Chapter 7

Evaluation

With respect to the first two requirements of this project—to improve the performance of Whiley programs, and to maintain as much as possible the semantics of the language—two evaluations have been performed. The first evaluation is a performance evaluation which focuses on the performance difference from executing portions of programs on a GPU rather than the CPU, whereas the second evaluation measures how well the new compiler retains the behaviour of the reference Whiley to JVM compiler. For the third requirement of this project—that the programmer does not need to be involved in the process of transforming from Whiley to OpenCL—an evaluation need not be performed, as the compiler has not modified the Whiley language in any way.

7.1 Evaluation Methodology

7.1.1 Performance Evaluation

To evaluate the performance benefit gained by using the Whiley to OpenCL project, a suite of benchmarks (Figure 7.2) is used to measure the time it takes generated OpenCL code to run on a GPU or CPU. These benchmarks have been created for this project, and reflect common computational problems which are able to be parallelised and executed on a GPU.

Two GPUs and one CPU are used for this evaluation, an AMD Radeon HD 6750M, a NVIDIA C1060 Tesla and an Intel Core i7-2720QM @ 2.20GHz. These two GPUs are hosted on machines with different CPUs and operating systems so this will have an effect on the results. Details of the test systems can be seen in Table 7.1.

Each benchmark is run 50 times with a warmup of two runs to exclude the cost of driver setup. The data is gathered inside the Java OpenCL runtime library, and timings are col-

Device	Host OS	Host CPU & Memory	GFLOPS
AMD Radeon HD 6750M 1GB 480 pipelines @ 600MHz	OS X 10.8.4	Intel Core i7-2720QM @ 2.20GHz 12GB	576
Intel Core i7-2720QM 12GB 4 cores @ 2.20GHz	OS X 10.8.4	Intel Core i7-2720QM @ 2.20GHz 12GB	70–106
NVIDIA C1060 Tesla 4GB 240 stream processors @ 1.3GHz	Ubuntu 12.04.3 LTS	Intel Xeon E5640 @ 2.67GHz 12GB	933

Table 7.1: The different systems used for the performance evaluation.

Name	Description	Optimised Portion
`gameoflife`	Conway's Game of Life on a 200x200 grid for three generations. Starting configuration is a single Pulsar [43] centred on the grid.	Each cell in the grid is computed by one work item. Each generation is a separate OpenCL dispatch.
`mandelbrot_float`	The computation of the Mandelbrot fractal using floating point arithmetic for a 1,000x1,000 pixel image. The maximum number of iterations per pixel is 1,024, and a bail out value of 4 is used.	The entire image is computed in one OpenCL dispatch, each pixel is computed by a different work item.
`mandelbrot_int`	The computation of the Mandelbrot fractal using integer arithmetic for a 1,000x1,000 pixel image. The maximum number of iterations per pixel is 1,024, and a bail out value of 4 is used.	The entire image is computed in one OpenCL dispatch, each pixel is computed by a different work item.
`gaussian_blur`	A 16 pixel blur on a 200x200 pixel image.	Each output pixel is computed by one work item.
`matrix_multiply`	The multiplication of two 300x300 matrixes.	Each cell in the resulting matrix is computed by one work item.
`n-body`	A simple n-body simulation with N = 300.	Each bodies' motion is computed by one OpenCL work item, each tick of the simulation is handled by a separate OpenCL dispatch.
`reduce_sum`	The summation of 524,288 elements using iterative reduction with a block size of 4 and a lower bound of 64 blocks.	Each block of 4 elements is summed by a single work item, each reduction step is a separate OpenCL dispatch.

Table 7.2: The seven different benchmarks used for the performance evaluation.

lected for total elapsed time, initialisation, marshalling, upload (minus marshalling as data is asynchronously uploaded during marshalling so the two cannot be measured independently), execution, download, unmarshalling and other overheads. Due to the fact that the OpenCL kernels are compiled and then cached for later reuse, the cost of the driver setup is not included in these measurements as it only occurs during the first warmup run.

7.1.2 Correctness Evaluation

To validate the correctness of the Whiley to OpenCL compiler a suite of unit tests is run and the results of executing these programs is compared against a specification. This suite of unit tests uses the existing Whiley test suite containing 412 tests, and a smaller set of 24 tests developed to cover edge cases specific to this compiler. Of the existing 412 tests for the Whiley compiler only a small portion contain loops which can be parallelised, meaning that many will not be run on the GPU. These 412 tests are used as they ensure that non-loop code and unsupported loops are correctly handled by the compiler, while the 24 new tests more specifically target loops which can run on the GPU.

7.1.3 Validity of not Comparing Whiley and OpenCL

Because the Whiley to OpenCL compiler changes the semantics of the Whiley language slightly, it is not possible to evaluate the performance of the compiler by comparing it to the existing Whiley to JVM compiler. The difference between these compilers because of using hardware arithmetic vs software arithmetic is so large that it inherently obscures the performance advantages or disadvantages of running code on a CPU or GPU.

To avoid this problem the compiler is used to generate OpenCL code, and this code is then compared when it runs on the GPU or the CPU. While this comparison is not representative of Whiley on the JVM vs Whiley running on a GPU, it does allow an investigation of the performance advantages from parallelisation and potential bottlenecks in the implementation.

In the future when Whiley is capable of utilising hardware arithmetic this comparison will be more characteristic of the true difference between Whiley on the JVM and Whiley running on a GPU, though marshalling and unmarshalling costs as well as the difference between the JVM and the OpenCL compiler will still affect the results.

7.2 Results and Discussion

7.2.1 Speed Improvements

The figures reported have been split into two different categories to illustrate the major differences in the way programs behave when run on the GPU. These two categories are the total time it took to execute the loops on the GPU—this includes initialisation, marshalling, upload, execution, download and unmarshalling—and the time it took just to execute. The main difference between these two categories is that the total time category also includes the overhead of getting the data needed to the GPU, whereas the execution time only shows how different the execution speeds of GPUs and CPUs are.

As can be seen from the results in Figure 7.1 in almost all cases the execution time on the GPU was substantially faster than running on the CPU, even if the same performance improvement is not seen in the total execution time—execution time for GPUs range from 0.18x to 31x the speed of the CPU and total times only range from 0.21x to 5.2x. Both GPUs are compared to the one CPU in these benchmarks, and different results are observed between the AMD Radeon HD 6750M and the NVIDIA C1060 Tesla as expected.

Across the seven benchmarks different results are seen in the speed improvements achieved by the compiler when executing on a GPU. Some benchmarks consistently show that running on the GPU provides a large performance gain while others perform poorly.

For three of the benchmarks—`mandelbrot_float`, `mandelbrot_int` and `gaussian_blur`—running on a GPU is substantially faster, while one of the benchmarks—`n-body`—is consistently slower.

For the `gameoflife` benchmark execution times on the GPU were significantly improved over the CPU, however total times were slower suggesting that the runtime overhead of calling out to the GPU is outweighing the performance gains in execution time.

The `matrix_multiply` benchmark sees significantly slower execution times when running on the GPU, however this doesn't significantly effect the total time of execution, implying that even when running on the CPU most of the time is being spent converting data formats rather than actually executing the loop body.

The `reduce_sum` benchmark exhibits the same behaviour, though the host system with the NVIDIA C1060 Tesla appears to be fast enough to overcome this and still record a speed improvement. This suggests that in this case the effect of the host machine is impacting the

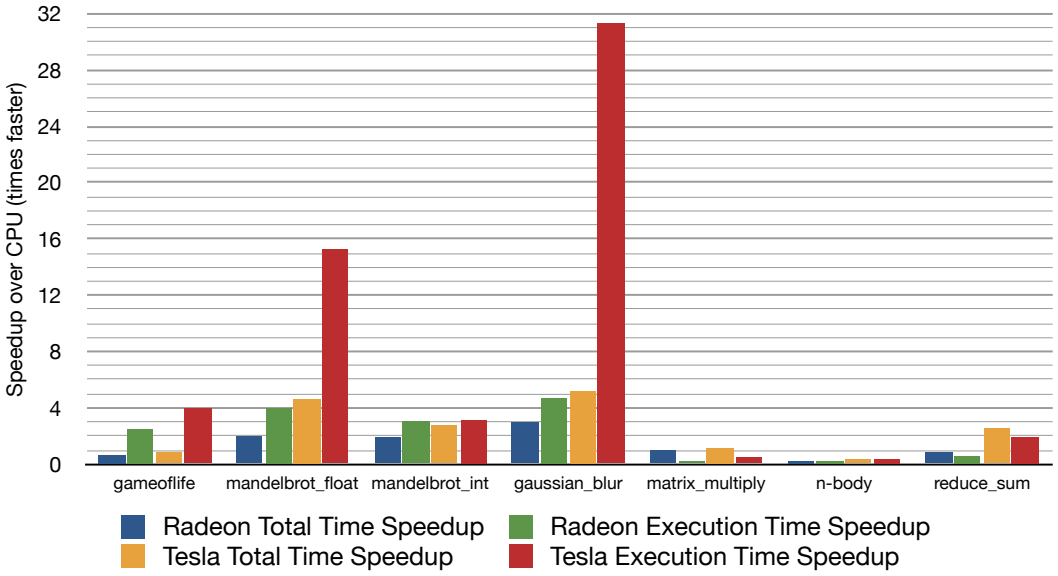


Figure 7.1: The relative speedups for total time and execution time. Each bar shows the speed improvement over the reference CPU. Speed improvements for both GPUs are split between the improvement in total execution time, and the improvement in execution time. Longer bars are faster.

results, as the time spent preparing data for upload is heavily dependent on the speed of the host CPU.

7.2.1.1 The `gameoflife` benchmark

For `gameoflife` the actual execution time was substantially improved when running on a GPU (2.5x faster for the AMD Radeon HD 6750M and 4.0x faster for the NVIDIA C1060 Tesla), however the total time was slower in both cases (0.66x as fast for the AMD Radeon HD 6750M and 0.82x for the NVIDIA C1060 Tesla). This suggests that execution time is not the bottleneck, instead either marshalling/unmarshalling, initialisation or upload-/download are the cause.

By looking at Figure 7.2 it is evident that marshalling + upload and download time (green + yellow and purple respectively) are the two areas which are significantly slower for the two GPUs than for the CPU. This points to the fact that for some programs the cost of getting data to/from the GPU provides a bottleneck in the process as it outweighs the gains made from faster loop execution.

The time it takes to upload and download data is entirely a factor of the amount of data transferred and the speed of the CPU-GPU bus. As there is no feasible way of improving the latter, the only way in which the transfer time could be reduced is by transferring less data. By looking at the `gameoflife` benchmark it becomes evident that this can actually be achieved. The `gameoflife` benchmark involves a large amount of redundant data transfer, and if this were to be removed then the benchmark would see significant improvements.

Each generation of the game of life is processed by reading from one dataset and writing to another, and then these are swapped. Each of these generations requires the entire process of initialisation, marshalling, upload, execution, download, unmarshalling to be repeated—three times for this benchmark as can be seen in Figure 7.2—even though this makes some of these steps redundant. For the `gameoflife` benchmark the two datasets do not need to be downloaded, swapped and re-uploaded, instead they could just be uploaded once, swapped

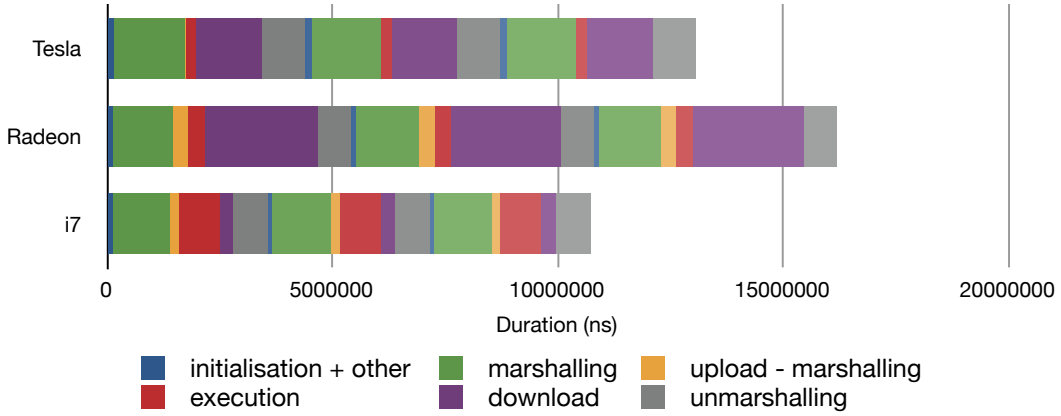


Figure 7.2: The duration of each part of loop execution for the `gameoflife` benchmark. Even though the execution times for the two GPUs is faster, the total time in both cases is slower. The GPUs are being slowed down by the transfer of data between the CPU and GPU. **Note:** The NVIDIA C1060 Tesla does appear to have an edge over the AMD Radeon HD 6750M in upload and download—this is likely to be partially caused by the differences in host machine.

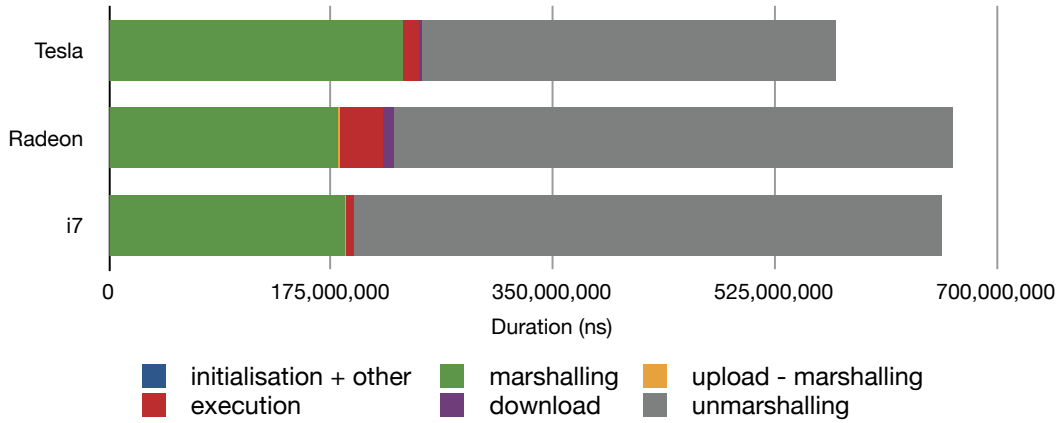


Figure 7.3: The duration of each part of loop execution for the `matrix_multiply` benchmark.

on the GPU and only downloaded once. In this benchmark this would remove two of three marshal/upload/download/unmarshal phases and this percentage only improves as the number of iterations increases.

This problem and solution are what lead Leung et al. [28] to add **Multi-pass loops** to the loop categorisation system as discussed in §2.5.1. These allow the compiler to detect this specific case and avoid the redundant copies.

7.2.1.2 The `matrix_multiply` benchmark

Matrix multiplication is typically an operation which parallelises very well and highlights the computation power available on GPUs [42, 5]. However in the matrix multiplication benchmark used for this evaluation execution on the GPU is actually slower than on the CPU.

An explanation for why these results differ so much from expected results can be found in several papers [42, 13, 38, 41]. Specifically a lecture from the University of Bristol [31]

illustrates the reason why this is. The slides from this lecture describe four different implementations of matrix multiplication with different memory access optimisations implemented on the GPU, and the speed improvements each implementation has over the CPU. For the two basic implementations the kernel running on the GPU was slower at only 0.33x and 0.67x as fast as the CPU, and to actually gain a speed improvement very specific optimisations need to be applied. Such optimisations can only be done though detailed knowledge of the structure of the program in order to make effective use of the memory hierarchy of GPUs. Compiler implementations do exist which are capable of performing these optimisations through deep program analysis, though the techniques used such as polyhedral code generation [41] are beyond the scope of this project.

This highlights how sensitive GPUs are to memory accesses, and the location of data within the memory hierarchy. In contrast as a programmer for CPUs these issues are typically obscured by layers of caches which automatically attempt to hide the details of this hierarchy from the programmer.

Another issue this benchmark emphasises is the cost of marshalling and unmarshalling. By looking at the amount of time spent performing each part of the loop (Figure 7.3) it can be seen that marshalling and unmarshalling times (green and grey) substantially outweigh execution time. This indicates that the cost of converting the Whiley types to OpenCL types is very expensive for this benchmark, and in this case is caused by the fact that the conversion from `real` to `float` is very computationally demanding.

In all programs the speedup which can be achieved by parallelisation and execution on the GPU is inherently limited by Amdahl's law [1]. This states that the maximum speedup from using multiple processors is limited by the fundamentally sequential component—in this case uploading and downloading data.

The effect of the differences in the host machine for the AMD Radeon HD 6750M and the NVIDIA C1060 Tesla can be strongly seen in these results as the marshalling phase is substantially slower on the Tesla's host CPU, and unmarshalling is much faster. This difference is what accounts for the better total time for the NVIDIA C1060 Tesla, rather than improved execution time.

7.2.1.3 Differences between the AMD Radeon HD 6750M and the NVIDIA C1060 Tesla

In all benchmarks the NVIDIA C1060 Tesla outperforms the AMD Radeon HD 6750M both for total time and execution time. This is expected as the AMD Radeon HD 6750M is a midrange laptop GPU while the NVIDIA C1060 Tesla is a dedicated GPGPU card. However the difference in performance between `mandelbrot_float` and `mandelbrot_int` is unexpected. For the floating point version of the Mandelbrot computation the NVIDIA C1060 Tesla has a 3.8x faster execution time than the AMD Radeon HD 6750M, but for the integer version the NVIDIA C1060 Tesla is only 1.03x as fast, even though the only alteration is the change from `float` to `int`.

This difference illustrates the variability in GPGPU performance across vendors and cards, and how sensitive GPU computation is to program structure and behaviour.

7.2.2 Scaling

To determine how problem size effects the performance of the OpenCL kernels a separate evaluation was performed. For this analysis the problem size was varied substantially and the results from these executions were collected. This evaluation only included two of the seven benchmarks, `mandelbrot_float` and `gaussian_blur`.

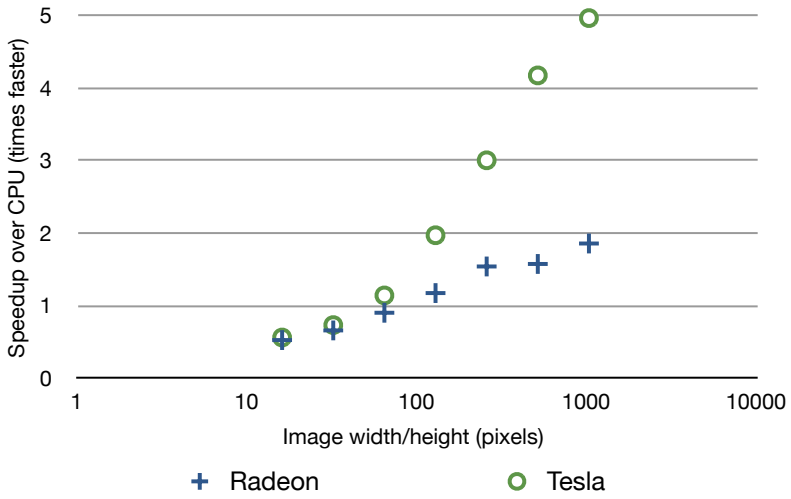


Figure 7.4: The relative speedup on the `mandelbrot_float` benchmark with increasing problem size. **Note** the logarithmic horizontal axis and how after an image size of 64x64 pixels both GPUs consistently outperform the CPU.

7.2.2.1 Scaling of `mandelbrot_float`

For the `mandelbrot_float` benchmark how the program performs as the problem size increases follows a clear pattern (Figure 7.4). For much smaller problem sizes execution on the CPU is faster, but after a certain problem size is reached the GPUs surpasses the CPU in terms of performance. In this case the relationship between the problem size and speedup is logarithmic, and the crossover point is near 64x64 pixels. As can be seen from Figure 7.4 the trend for the NVIDIA C1060 Tesla is steeper than for the AMD Radeon HD 6750M, indicating that the NVIDIA C1060 Tesla is the more powerful card for larger problems by a constant factor, however in both cases either GPU is faster than the CPU.

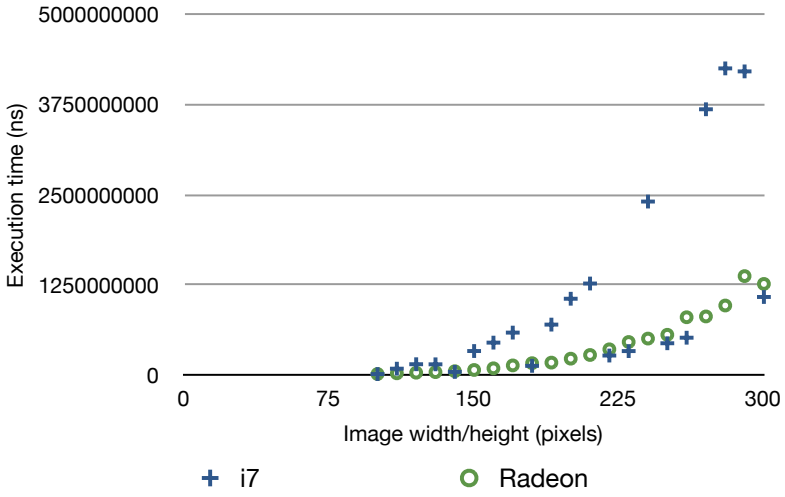
This is indicative of the fact that GPUs are better suited to solving particular problems and executing programs with certain structures, however this comes with a cost.

7.2.2.2 Scaling of `gaussian_blur`

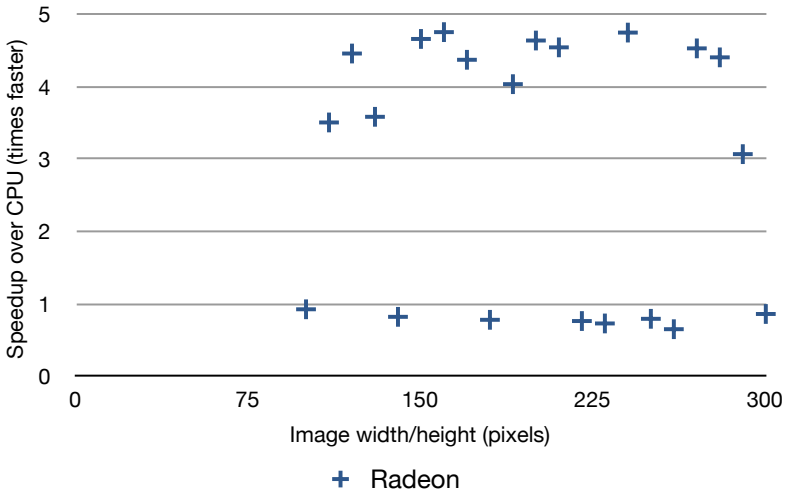
The `gaussian_blur` benchmark does not exhibit the smooth trends seen with `mandelbrot_float`, instead the speedup observed is highly dependent on the problem size in an unpredictable manner (see Figure 7.5b). These results suggest that either the CPU or the GPU is highly sensitive to the problem size, most likely due to memory access patterns or cache behaviour.

To illustrate the problem further the raw execution times are shown in Figure 7.5a, which shows that while the GPU exhibits a nice consistent slowdown as the problem size grows, the CPU inconsistently performs at around the same speed of the GPU, or approximately 4x slower.

The majority of the time the CPU is significantly slower, however for certain problem sizes it is on par with the GPU, suggesting that the way in which the CPU caches are being used is the cause of the observed effect [27]. However further data will need to be collected from the CPU performance counters to be able to validate whether this is the case.



(a) The raw execution times for CPU and GPU.



(b) The relative speedups.

Figure 7.5: The speedup and execution time on the `gaussian_blur` benchmark with increasing problem size. **Note** how sensitive and inconsistent the speedup and CPU execution time are with respect to the problem size. For these two figures only results from the AMD Radeon HD 6750M and Intel Core i7-2720QM @ 2.20GHz are shown as the NVIDIA C1060 Tesla needlessly complicates the plots.

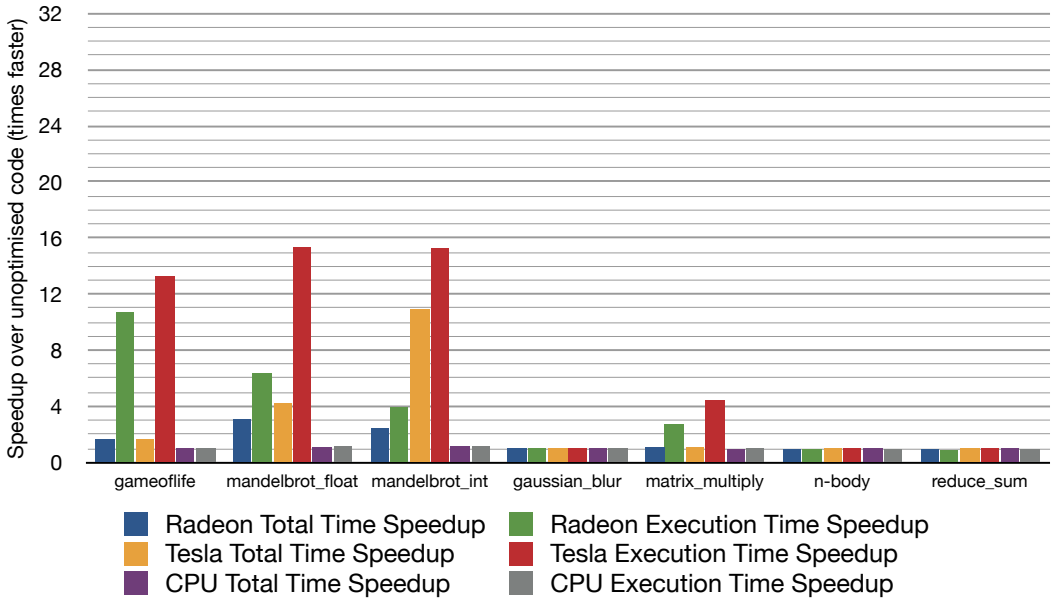


Figure 7.6: The performance gains from multidimensional loop flattening.

7.2.3 Optimisations

To investigate how optimisations impact the performance of the OpenCL kernels the seven different benchmarks were run with optimisations on and off. The only optimisation that is currently capable of being disabled is multidimensional loop flattening (§6.2.1), so the results below are presented with only this enabled and disabled.

As can be seen in Figure 7.6, the multidimensional loop flattening optimisation has a large impact on the performance of certain benchmarks. In four of the seven benchmarks performance is improved, the largest of these performance improvements is 11x total time, and 15x execution time with the optimisation. For the remaining three benchmarks no performance change is seen, and this is expected as these benchmarks do not have loops which are compatible with the multidimensional loop flattening optimisation.

For the benchmarks which do exhibit a large speedup, this optimisation allows the compression of two nested loops into one multidimensional loop, meaning that K work items executing a loop body J times turns into $K \times J$ work items executing the loop body once. This takes advantage of the edge GPUs have over CPUs with large numbers of threads, and substantially improves the performance of the OpenCL kernels.

For the `matrix_multiply` benchmark this optimisation changes the number of threads from 300 to 90,000, and for `gameoflife` from 200 to 40,000. The improvement is even more marked with the Mandelbrot benchmarks which go from 1,000 threads to 1,000,000. This is reflected in the results, with these two benchmarks gaining more performance than those with a smaller increase in the number of threads.

As this optimisation largely favours GPUs very little difference in performance is seen with the kernels executing on the CPU. The greatest gain the CPU sees is with the `mandebrot_int` benchmark which improves the total time by 1.16x, and execution time 1.20x.

In no cases is the performance with this optimisation enabled slower, which validates the decision to apply this by default to all possible loops.

7.2.4 Correctness

Of the 24 tests specifically created for this compiler all 24 pass, ensuring that the compiler produces programs which do perform correctly when run on the GPU, and when encountering several edge cases. When executed on the entire suite of 412 existing unit tests for the Whiley compiler only 95.4% (393) execute without issues. However this is to be expected as several of the tests include Whiley language features which are not yet supported by the compiler. Of the 19 failing tests all 19 fail for known reasons:

- 1 test fails as Whiley reference counting is incorrectly handled in some cases,
- 4 tests fail because of an incompatible jar file version,
- 2 tests fail because lambdas in Whiley are currently not supported,
- 1 test fails because Whiley's dead code elimination leaves behind an empty method body,
- 5 tests fails because processes in Whiley are currently not supported,
- 2 tests currently fail as set comprehensions are not supported,
- 4 tests fail because exceptions are not supported.

This result strongly suggests that with the exception of programs containing certain unimplemented language features, the compiler is capable of producing code which does not change the semantics of the Whiley programming language to a significant degree.

Chapter 8

Future Work

The primary focus for future work is to support more Whiley language features, and allow a greater range of Whiley programs to be compatible with the compiler and executing on GPUs. A large amount of research has been performed in areas relevant to this project that could provide opportunities to improve performance. Investigating these areas further and implementing these algorithms for the Whiley compiler is a possible future line of development.

8.1 Whiley Language Features

As seen from the correctness evaluation (§7.2.4) the Whiley to OpenCL compiler is currently not capable of processing programs which contain certain language features. For example exceptions and lambdas cause the compiler to crash, and call-by-reference semantics are not maintained by the conversion to OpenCL. All these issues can be resolved by improvements to the program analyser, and once these are implemented the compiler will be able to handle all Whiley programs, even if they cannot be run on the GPU.

8.1.1 Extending Support for Datatypes on the GPU

Beyond this the compiler can also be extended to support additional datatypes. These include maps, records and tuples of varying types, as well as the possibility of implementing unions. Adding these types requires addressing issues of alignment, packing and in-memory representations, as the Whiley implementations of these types make heavy use of references, polymorphism and RunTime Type Identification (RTTI), all of which cannot be used when uploading data to the GPU.

8.1.2 Arbitrary Precision Arithmetic

The Whiley language is designed to feature extended static checking to ensure that programs run correctly. As the language matures there are intentions to implement methods for verifying the bounds of numeric types (such as [12, 15]) so that hardware arithmetic can be used in cases where the bounds of the values are within the bounds of the hardware types, and the resultant error from the computations is within desired tolerances. When the Whiley language is capable of performing this analysis the Whiley to OpenCL compiler will be able to integrate this information to determine whether loops can be executed on the GPU without invalidating these constraints. In addition if hardware types are used by the Whiley language then the processes of marshalling and unmarshalling will become much simpler, further improving the performance of the program.

8.2 Performance

The performance of the Whiley to OpenCL compiler can be improved significantly. Many techniques exist which can allow the compiler to more efficiently exploit the resources of GPUs [42, 18, 29, 6], especially by improving data transfers [17, 21] and the utilisation of the GPU memory hierarchy [4, 16, 14, 41].

One simple improvement which is has not been completed in the compiler is the ability to determine which parameters to a kernel are going to be read-only, as these need not be downloaded from the GPU because they will not have been modified.

Another important aspect to improving the performance of compiled programs is utilising heuristics to accurately determine whether loops will actually benefit from execution on the GPU, as for some programs the differences in performance characteristics will result in degraded performance rather than improvements.

8.3 Extending Device and Language Support

Exploring the behaviour of this compiler with different hardware supported by OpenCL such as FPGAs is a logical extension of this project, and beyond this other combinations of runtime languages and platform specific languages such as CUDA.

Chapter 9

Conclusions

This project has successfully extended the Whiley compiler to allow Whiley programs to be partially executed on a GPU. Two optimisations have been developed which improve the performance of these programs when running on the GPU.

The first of these optimisations reduces the amount of data which needs to be transferred to and from the GPU by identifying loops which iterate over ranges of numbers and computing these numbers on the GPU rather than the CPU. The second optimisation flattens nested loops into multidimensional loops whenever possible, which allows the compiler to exploit these and generate code which uses a greater number of threads.

The optimisations and additions to the compiler have been evaluated and show that the performance of Whiley programs is improved as intended, while still preserving much of the semantics of Whiley programs. Overall results show up to a 5.2x total speed improvement, and a 31x execution time improvement, while the multidimensional loop flattening optimisation alone provided 11x total time and 15x execution time improvements. Several problems with preserving Whiley semantics have been identified, and the current restrictions of this approach have been detailed.

Future avenues for producing more performant code have been identified, as well as ways in which the Whiley programming language can be modified to assist in the creation of programs which execute faster, and adhere even closer to the intended semantics of Whiley.

Overall the developed compiler is capable of providing performance gains to many Whiley programs, though further work is needed to extend the scope of programs which are able to benefit from running on GPUs.

Bibliography

[1] AMDAHL, G. M. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*. AFIPS '67 (Spring). ACM, New York, NY, USA, 483–485.

[2] APPLE INC. objc.h. `http://www.opensource.apple.com/source/objc4/objc4-371.1/runtime/objc.h`.

[3] ARTHO, C., HAVELUND, K., AND BIERE, A. 2003. High-level data races. *Software Testing, Verification and Reliability 13*, 4, 207–227.

[4] ARTIGAS, P. V., GUPTA, M., MIDKIFF, S. P., AND MOREIRA, J. E. 2000. Automatic loop transformations and parallelization for Java. In *Proceedings of the 14th international conference on Supercomputing*. ICS '00. ACM, New York, NY, USA, 1–10.

[5] BECK, R., LARSEN, H. W., AND JENSEN, T. 2011. Extending scala with general purpose GPU programming. Student report, Aalborg University, Department of Computer Science. July.

[6] BETTS, A., CHONG, N., DONALDSON, A., QADEER, S., AND THOMSON, P. 2012. GPUVerify: a verifier for GPU kernels. *SIGPLAN Not. 47*, 10 (Oct.), 113–132.

[7] BLYTHE, D. 2006. The direct3d 10 system. In *ACM SIGGRAPH 2006 Papers*. SIGGRAPH '06. ACM, New York, NY, USA, 724–734.

[8] CALVERT, P. 2010. Parallelisation of Java for graphics processors. *Final-year dissertation at University of Cambridge Computer Laboratory. Available from http://www. cl. cam. ac. uk/ prc33*.

[9] CHALABINE, M. 2006. Invasive interactive parallelization. In *Proceedings of the Doctorial Symposium of the Fourteenth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE 14), Portland, Oregon, USA*.

[10] CHAN, B. AND ABDELRAHMAN, T. S. 2004. Run-time support for the automatic parallelization of Java programs. *Journal of Supercomputing* 28, 91–117.

[11] DEAN, J. AND GHEMAWAT, S. 2008. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1, 107–113.

[12] D'SILVA, V., HALLER, L., KROENING, D., AND TAUTSCHNIG, M. 2012. Numeric bounds analysis with conflict-driven learning. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 48–63.

[13] DU, P., WEBER, R., LUSZCZEK, P., TOMOV, S., PETERSON, G., AND DONGARRA, J. 2012. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Comput. 38*, 8 (Aug.), 391–407.

[14] DUBACH, C., CHENG, P., RABBAH, R., BACON, D. F., AND FINK, S. J. 2012. Compiling a high-level language for GPUs (via language support for architectures and compilers). *SIGPLAN Not. 47,* 6 (June), 1–12.

[15] GOUBAULT, E. AND PUTOT, S. 2006. Static analysis of numerical algorithms. In *Static Analysis*. Springer, 18–34.

[16] GROSSER, T., ZHENG, H., ALOOR, R., SIMBÜRGER, A., GRÖSSLINGER, A., AND POUCHET, L.-N. 2011. Polly—polyhedral optimization in llvm. In *First International Workshop on Polyhedral Compilation Techniques*. IMPACT, 1–6.

[17] GUO, J., THIYAGALINGAM, J., AND SCHOLZ, S.-B. 2011. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. DAMP '11. ACM, New York, NY, USA, 15–24.

[18] HAN, T. D. AND ABDELRAHMAN, T. S. 2009. hiCUDA: a high-level directive-based language for GPU programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. GPGPU-2. ACM, New York, NY, USA, 52–61.

[19] IEEE AND THE OPEN GROUP. <stdbool.h>. `http://pubs.opengroup.org/onlinepubs/007904875/basedefs/stdbool.h.html`.

[20] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION AND INTERNATIONAL ELECTROTECHNICAL COMMISSION AND OTHERS. Iso/iec 9899: 1999.

[21] JABLIN, T. B., PRABHU, P., JABLIN, J. A., JOHNSON, N. P., BEARD, S. R., AND AUGUST, D. I. 2011. Automatic CPU-GPU communication management and optimization. *SIGPLAN Not. 47,* 6 (June), 142–151.

[22] JOCL. Java bindings for OpenCL. `http://jocl.org`.

[23] KASAHARA, H., OBATA, M., AND ISHIZAKA, K. 2001. Automatic coarse grain task parallel processing on smp using openmp. In *Languages and Compilers for Parallel Computing*. Springer, 189–207.

[24] KHRONOS OPENCL WORKING GROUP. OpenCL - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/` accessed: 19 March 2013.

[25] KHRONOS OPENCL WORKING GROUP. The OpenCL specification. `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`.

[26] KHRONOS OPENCL WORKING GROUP. OpenGL - the industry standard for high performance graphics. `http://www.opengl.org`.

[27] LEE, V. W., KIM, C., CHHUGANI, J., DEISHER, M., KIM, D., NGUYEN, A. D., SATISH, N., SMELYANSKIY, M., CHENNUPATY, S., HAMMARLUND, P., ET AL. 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ACM SIGARCH Computer Architecture News*. Vol. 38. ACM, 451–460.

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

[29] LI, P., LI, G., AND GOPALAKRISHNAN, G. 2012. Parametric flows: automated behavior equivalencing for symbolic analysis of races in CUDA programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. IEEE Computer Society Press, Los Alamitos, CA, USA, 29:1–29:10.

[30] LUEBKE, D., HARRIS, M., KRÜGER, J., PURCELL, T., GOVINDARAJU, N., BUCK, I., WOOLLEY, C., AND LEFOHN, A. 2004. GPGPU: general purpose computation on graphics hardware. In *ACM SIGGRAPH 2004 Course Notes*. SIGGRAPH '04. ACM, New York, NY, USA.

[31] MCINTOSH-SMITH, S. Lecture 3: Optimising OpenCL performance. `http://www.cs.bris.ac.uk/home/simonm/workshops/OpenCL_lecture3.pdf`.

[32] NVIDIA CORPORATION. The CUDA specification. `http://docs.nvidia.com/cuda/index.html` accessed: 19 March 2013.

[33] NYSTROM, N., WHITE, D., AND DAS, K. 2011. Firepile: run-time compilation for GPUs in scala. *SIGPLAN Not. 47,* 3 (Oct.), 107–116.

[34] ORCHARD, D. A., BOLINGBROKE, M., AND MYCROFT, A. 2010. Ypnos: declarative, parallel structured grid programming. In *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*. ACM, 15–24.

[35] OWENS, J. D., HOUSTON, M., LUEBKE, D., GREEN, S., STONE, J. E., AND PHILLIPS, J. C. 2008. GPU computing. *Proceedings of the IEEE 96,* 5, 879–899.

[36] PEARCE, D. J. Whiley: an open source programming language with extended static checking. `http://whiley.org` accessed: 16 March 2013.

[37] PEARCE, D. J. AND NOBLE, J. 2011. Implementing a language with flow-sensitive and structural typing on the JVM. *Electronic Notes in Theoretical Computer Science 279,* 1, 47–59.

[38] PUŹNIAKOWSKI, T. 2012. Performance of OpenCL.

[39] RAUCHWERGER, L., AMATO, N. M., AND PADUA, D. A. 1995. Run-time methods for parallelizing partially parallel loops. In *Proceedings of the 9th international conference on Supercomputing*. ACM, 137–146.

[40] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS) 15,* 4, 391–411.

[41] VERDOOLAEGE, S., CARLOS JUEGA, J., COHEN, A., IGNACIO GÓMEZ, J., TENLLADO, C., AND CATTHOOR, F. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim. 9,* 4 (Jan.), 54:1–54:23.

[42] VOLKOV, V. AND DEMMEL, J. W. 2008. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. SC '08. IEEE Press, Piscataway, NJ, USA, 31:1–31:11.

[43] WEISSTEIN, E. W. Pulsar from Eric Weisstein's Treasure Trove of Life C.A. `http://www.ericweisstein.com/encyclopedias/life/Pulsar.html`.