Accelerating Interpreted Programming Languages on GPUs with Just-In-Time Compilation and Runtime Optimisations

Juan José Fumero Alfonso

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2017

Abstract

Nowadays, most computer systems are equipped with powerful parallel devices such as Graphics Processing Units (GPUs). They are present in almost every computer system including mobile devices, tablets, desktop computers and servers. These parallel systems have unlocked the possibility for many scientists and companies to process significant amounts of data in shorter time. But the usage of these parallel systems is very challenging due to their programming complexity. The most common programming languages for GPUs, such as OpenCL and CUDA, are created for expert programmers, where developers are required to know hardware details to use GPUs.

However, many users of heterogeneous and parallel hardware, such as economists, biologists, physicists or psychologists, are not necessarily expert GPU programmers. They have the need to speed up their applications, which are often written in high-level and dynamic programming languages, such as Java, R or Python. Little work has been done to generate GPU code automatically from these high-level interpreted and dynamic programming languages. This thesis presents a combination of a programming interface and a set of compiler techniques which enable an automatic translation of a subset of Java and R programs into OpenCL to execute on a GPU. The goal is to reduce the programmability and usability gaps between interpreted programming languages and GPUs.

The first contribution is an Application Programming Interface (API) for programming heterogeneous and multi-core systems. This API combines ideas from functional programming and algorithmic skeletons to compose and reuse parallel operations.

The second contribution is a new OpenCL Just-In-Time (JIT) compiler that automatically translates a subset of the Java bytecode to GPU code. This is combined with a new runtime system that optimises the data management and avoids data transformations between Java and OpenCL. This OpenCL framework and the runtime system achieve speedups of up to 645x compared to Java within 23% slowdown compared to the handwritten native OpenCL code.

The third contribution is a new OpenCL JIT compiler for dynamic and interpreted programming languages. While the R language is used in this thesis, the developed techniques are generic for dynamic languages. This JIT compiler uniquely combines a set of existing compiler techniques, such as specialisation and partial evaluation, for OpenCL compilation together with an optimising runtime that compile and execute R code on GPUs. This JIT compiler for the R language achieves speedups of up to 1300x compared to GNU-R and 1.8x slowdown compared to native OpenCL.

Lay Summary of Thesis

Nowadays, most computer systems are equipped with powerful parallel devices such as Graphics Processing Units (GPUs). They are present in almost every computer system including mobile devices, tablets, desktop computers and servers. These parallel systems have unlocked the possibility for many scientists and companies to process significant amounts of data in shorter time. But the usage of these parallel systems is very challenging due to their programming complexity. The most common programming languages for GPUs, such as OpenCL and CUDA, are created for expert programmers, where developers are required to know hardware details to use GPUs.

However, many users of heterogeneous and parallel hardware, such as economists, biologists, physicists or psychologists, are not necessarily expert GPU programmers. They have the need to speed up their applications, which are often written in high-level and dynamic programming languages such as Java, R or Python. Little work has been done to generate GPU code automatically from these high-level interpreted and dynamic programming languages.

This thesis presents a combination of a novel programming interface and a set of compiler techniques which enable an automatic translation of a subset of Java and R programs into GPU code. At the same time, it offers comparable performance compared to native implementations.

Acknowledgements

Many people have contributed in one way or another during my PhD. Firstly, I would like to thank my supervisors, Christophe Dubach and Michel Steuwer for the great supervision and help during my PhD.

Secondly, I would like to thank the Oracle Labs compiler group in Linz, Austria, for the constant technical support. In particular Thomas Würthinger, Lukas Stadler, Roland Schatz, Andreas Wöß, Gilles Duboscq, Christian Wirth, Stefan Marr, Josef Eisl, Manuel Rigger, David Leopoldseder, Matthias Grimmer and the Italian guys, Daniele Bonetta and Danilo Ansaloni. It was great to work with them during my internship, where one of the main ideas of this thesis was born.

I would like to thank Sebastian Ertel and Andres Goens from the University of Dresden for the feedback and testing one of the contributions presented in this thesis. I want to thank Murray Cole and Michael O'Boyle for the great feedback and discussions during the review panel meetings, as well as my colleagues Thibaut Lutz, Erik Tomusk and Yuan Wen and my friends Rosa Filgueira, Michael Rovatsos, José Cano, Alberto Magni, Chris Margiolas, Chris Vasiladiotis and Dimitrios Papadopoulos. I am also grateful to have discussed much of the material presented in this thesis with David Kaeli and Christos Kotselidis.

I would also like to thank my parents, Eva and Armando, and my sisters, Carmen and Anabel, my source of inspiration. I do not have enough words to thank Ruymán and Yurena for their constant presence and for hosting me in Edinburgh like their own child. Finally, I am grateful to Anna for her unconditional support and comprehension.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Some of the material used in this thesis has been published in the following papers:

- Juan J. Fumero, Michel Steuwer, Christophe Dubach. *A Composable Array Function Interface for Heterogeneous Computing in Java.* In Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, June 2014.

- Juan J. Fumero, Toomas Remmelg, Michel Steuwer, Christophe Dubach. *Runtime Code Generation and Data Management for Heterogeneous Computing in Java.* In Proceedings of ACM SIGPLAN International Conference on Principles and Practices of Programming on the Java Platform, September 2015.

- Juan Fumero, Michel Steuwer, Lukas Stadler, Christophe Dubach. *OpenCL JIT Compilation for Dynamic Programming Languages.* MoreVMs Workshop on Modern Language Runtimes, Ecosystems, and VMs, April 2017.

- Juan Fumero, Michel Steuwer, Lukas Stadler, Christophe Dubach. *Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation.* In Proceedings of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), April 2017.

[illegible]

(*Juan José Fumero Alfonso*)

Contents

List Of Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General-Purpose Graphics Processing Unit
Graal IR	Graal Intermediate Representation
IR	Intermediate Representation
JIT	Just-In-Time (referring to compilation)
JVM	Java Virtual Machine
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SSA	Static Single Assignment
UDF	User Defined Function
VM	Virtual Machine

Chapter 1

Introduction

Nowadays, most computer systems are equipped with powerful parallel devices, such as Graphics Processing Units (GPUs). They are present in almost every computer system including mobile devices, tablets, desktop computers and servers in data centres. For example, one of the most powerful supercomputers in the world includes more than eighteen thousand GPUs, (the Titan supercomputer [TOP 17]). For a few hundred pounds people can acquire a GPU that contains hundreds of computing cores. GPUs are powerful devices that have unlocked the possibility for many companies and scientists to manipulate huge amounts of data faster than traditional CPUs. Besides, GPUs are much cheaper and consume much less energy than the equivalent CPU systems.

However, the significant speedups that users obtain when running applications on GPUs are at the cost of complex programmability. GPUs are a completely different type of hardware compared to traditional CPUs. To use and optimise GPU applications, programmers must know details about the GPU architecture and GPU parallel programming models. Programming languages such as OpenCL and CUDA facilitate the programmability of GPUs, leveraging hardware abstractions to programming languages like C. Yet, these programming languages target expert programmers, as the OpenCL standard explicitly says [OpenCL 09]:

"The target of OpenCL is expert programmers wanting to write portable yet efficient code. OpenCL provides a low-level hardware abstraction plus a framework to support programming and many [other] details of the underlying hardware."

However, many of the potential users of GPUs are non-experts in GPU and parallel programming. We live in the era of big data, where scientists and companies gather data from many sources, such as social media, and analyse them in large parallel computer systems and high performance computers because of significant amounts of data to be processed.

Many of the potential users of GPUs are economists, biologists, physicists, neuroscientists or physiologists. They have the software and the need for parallel execution using specialised hardware like GPUs for speeding up applications. However, these users tend to program in higher-level programming languages than C. They usually develop applications using managed and interpreted languages, such as Java and C# or even using interpreted and dynamic programming languages, such as Python, Ruby, R or Matlab.

Managed and interpreted programming languages offer portability and security and high-level programming features. Programs are executed in a virtual machine (VM) that loads the code to specific runtime areas in the VM, performs checks for security and manages the memory automatically. Interpreted and dynamic programming languages offer high-level functionality and simplicity of use, and the interpreter enables fast iterative software development. They are also executed in a VM that provides similar features to managed and interpreted languages.

1.1 The Problem

High-level programming languages have two main downsides. Firstly, programmers that use these programming languages are not necessarily experts of GPU programming. There are a lot of libraries and programming frameworks for Java, Python or R to use GPUs. These libraries and frameworks either provide a fixed number of operations that users can execute on GPUs, or they provide a wrapper that abstracts the GPU hardware complexity to the high-level programming model. In either case, users are required to firstly change the source code, and secondly to know low-level details, not only about the GPU parallel programming model but also about the GPU architecture. Besides, even using specific libraries or wrappers, good performance is not guaranteed. Moreover, when programming GPUs from high-level languages, not only the GPU code is important for performance, but also optimising data transformations between high-level and low-level languages is a critical task in order to achieve good performance.

Secondly, applications written in high-level programming languages are usually very slow compared to C (which is widely used in supercomputing). To increase performance, many implementation of these high-level programming languages make use of JIT compilers. JIT compilation is a technique to compile, at runtime, the most frequently executed parts of the program into efficient machine code using profiling information. However, when compiling dynamic and interpreted languages on GPUs, the JIT compilation process is even more complicated. Ideally, interpreters for dynamic programming languages would be able to exploit the GPU automatically and transparently. A possible solution is to port the interpreter logic to the GPU and directly execute the interpreted program on the GPU itself. However, this is not a good solution because part of the interpreter is complex and non-practical to port, such as method dispatch and dynamic memory allocation.

Little research has been done in the area of how to automatically use GPUs and heterogeneous hardware from high-level programming languages such as Java, Python or R, which are widely used nowadays. This thesis is focused on closing the gap between the heterogeneous hardware and interpreted and dynamic programming languages. This thesis first exploits how to program, compile at runtime and execute Java applications on GPUs and then it extends the presented techniques for the R interpreted and dynamically typed language.

1.2 The Solution

This thesis proposes a combination of compiler techniques to leverage the complexity of programming and tuning GPU applications to a JIT compiler and a runtime system. When the input application is running, the JIT compiler compiles the most frequently executed parts of the program into OpenCL C for targeting the GPU. The main goal of this thesis is to compile and execute applications written in Java and R into the GPU using OpenCL. To achieve this goal, the JIT compiler and the runtime are decomposed into smaller components.

Firstly, an Application Programming Interface (API) is proposed. This API is focused on array programming and uses well-known structured parallel skeletons, such as a map or reduce operations, to facilitate the programmability and the code generation. This API is hardware agnostic, which means that the way of programming is entirely independent of the hardware underneath.

Secondly, it extends an existing Java compiler framework to compile, at runtime, Java bytecode to OpenCL C. This compiler framework is called Graal. Graal is a Java optimising compiler developed at Oracle Labs to optimise and compile Java bytecode to x86 and Sparc architectures. Graal replaces the JIT compiler available in Java

HotSpot VM. This thesis extends the Graal compiler for GPU code generation via OpenCL.

Lastly, a new JIT compiler for dynamic and interpreted programming languages is proposed. The R language is used in this thesis as one of the main dynamic and interpreted programming languages adopted nowadays by industry and academia for data analytics and large data processing. The solution combines techniques at different levels of the compiler framework. First, it specialises the Abstract Syntax Tree (AST) for the OpenCL compilation using the Truffle framework, a domain specific language and an engine for implementing programming languages on top of the Java Virtual Machine (JVM). Secondly, it extends the existing partial evaluator in the Graal compiler (a process that receives an AST and produces an intermediate representation in a control flow graph form) for OpenCL. Partial evaluation is a well-known compiler technique to improve performance on CPUs, in which the input program is specialised at runtime. Then, it extends the compiler optimisations of Graal for OpenCL using new compiler phases that simplify the control flow graph before generating the OpenCL C code. As a result, most of the interpreter code is compiled away, leaving only the actual application logic and computation. However, because Graal is an optimising compiler, it makes speculations of what is being compiled during partial evaluation. As a consequence, the presented solution in this thesis includes a mechanism for dealing with miss-speculations on the GPU.

In summary, the last contribution of this thesis extends the existing compiler techniques that combine specialisation and partial evaluation, in which the language interpreter is specialised to the input program, to compile and execute user applications implemented in dynamically typed languages on GPUs.

1.3 Contributions

The thesis makes the following contributions.

Firstly, it presents a new Application Programming Interface (API) called JPAI (Java Parallel Array Interface) for array programming, parallel and heterogeneous computing within Java. JPAI uses pattern composition and function reusability to facilitate programming for multi-core CPUs and OpenCL for parallel execution on CPUs and GPUs. The main target of JPAI is for GPU programming, although it can be also used for multi-core execution. The computation expressed with JPAI is written once and executed everywhere. This means that with the same API, the program is written once and it can be executed on a multi-core CPU using Java threads or on a GPU using OpenCL transparently. The runtime explores the hardware available and rewrites the input program to be computed in parallel in the target architecture. Applications written with JPAI show that is possible to achieve speedups of up to 5x in a system with 4 cores with hyper-threading.

Secondly, it presents Marawacc, a compiler framework for OpenCL Just-In-Time (JIT) that automatically compiles a subset of the Java bytecode into OpenCL, and a runtime system that orchestrates the execution of the GPU program within Java. It uses the proposed JPAI to select the parallel pattern to be generated to OpenCL and it obtains the Java code of the user computation to be offloaded to the GPU. This framework also integrates a runtime system that optimises and executes the OpenCL kernel efficiently. One of the most important aspects to make heterogeneous hardware usable from managed programming languages is to optimise the data transformation between the high and low-level languages. This process is called marshalling. This thesis also presents a technique to avoid the marshalling and improve the overall performance in most cases. The experimental evaluation shows that Marawacc is, on average, 23% slower compared to OpenCL C++, 55% faster than existing GPU frameworks for Java and achieves speedups of up to 645x over sequential Java code when using a GPU.

Lastly, it presents an OpenCL JIT compiler for offloading part of the computation of the dynamic and interpreted R programs to the GPU automatically. It also presents a set of data optimisations that help to reduce data transfers between R and OpenCL. Interpreted and dynamic languages are normally very flexible in terms of typing where types can change at any point of the execution time. However, when compiling an interpreted program, an optimising compiler makes use of speculative assumptions about types. These assumptions may be invalidated at runtime if they are wrongly speculated. This thesis also presents a technique for dealing with miss-assumptions on GPUs and falls back to the interpreter to recover from miss-speculations (called deoptimisations). The performance evaluation shows that, by using the JIT compiler and executing on a GPU, R programs obtain up to 1300x speedup compared to GNU-R and only 1.8x slowdown compared to the optimised OpenCL C++.

These contributions help to reduce the gap between managed, dynamic and interpreted programming languages and GPUs, achieving performance close to native implementations.

1.4 Thesis Outline

The remaining of this thesis is organised as follows:

Part I: State of the Art

Chapter 2: this chapter presents the technical background of all the concepts that are needed to understand the rest of this work. It first introduces heterogeneous systems and gives an overview of how a GPU executes a program. Then it introduces algorithmic skeletons and it reviews the most relevant ones. The last section explains how managed languages work and, particularly, it explains how the Java Virtual Machine executes and compiles, at runtime, Java bytecode into efficient machine code.

Chapter 3: this chapter discusses the most relevant related works for heterogeneous programming on GPUs. It is focused on the related projects for just-in-time compilation for Java and the Java Virtual Machine. It also discusses the most related projects for dynamic and interpreted programming languages.

Part II: Contributions

Chapter 4: this chapter presents JPAI, a reusable and composable API for heterogeneous computing in Java. It describes the API and explains how computation can be easily expressed and composed. It also shows how scientific applications can be implemented in Java using JPAI. This chapter presents an evaluation of the API using Java threads for targeting multi-core CPUs. Part of the work presented in this chapter has been published in [Fume 14].

Chapter 5: this chapter presents how JPAI can be used to compile and execute Java programs at runtime to OpenCL C for executing on GPUs. It presents Marawacc, an OpenCL JIT compiler and a runtime system that orchestrates and optimises the GPU execution within Java. Finally, it shows a performance evaluation for a set of Java applications that are implemented using JPAI and executed on GPUs using the Marawacc framework. Part of the material presented in this chapter has been published in [Fume 15].

Chapter 6: this chapter presents a set of compiler techniques to automatically compile and execute R programs on GPUs. It shows a prototype for offloading, at runtime, part of the R program that are identified by the VM as hot functions, to the GPU via OpenCL. It also presents a runtime system that is able to optimise, execute and perform deoptimisations. Part of the work presented in this chapter has been published in [Fume 17a, Fume 17b].

Part III: Conclusions

Chapter 7: this chapter concludes the thesis and summarises the main contributions. It also discusses the main limitations of the proposed approach and suggests possible solutions to overcome each constraint. Finally, it proposes and discusses future work.

Part I

STATE OF THE ART

Chapter 2

Background

This chapter presents the background material and the main concepts related to parallel and heterogeneous computing that are required to understand the rest of this thesis. Section 2.1 shows an overview of heterogeneous computer systems. Section 2.2 explains how to program CPUs and GPUs using the OpenCL programming model. As this section will show, programming heterogeneous architectures is not an easy task since the programming abstractions for accelerators are still very low-level. Ideally, programmers should use high-level abstractions and rely on runtime and compiler optimisations to achieve high performance. Section 2.3 introduces parallel skeletons as an easier way of structuring and programming parallel applications for heterogeneous computing. This thesis is focused on implementing compiler techniques for heterogeneous computing within managed languages, and, in particular, from Java. Section 2.4 introduces the main concepts related to Java and the Java Virtual Machine (JVM) as well as the compiler infrastructure used in this thesis. Section 2.5 concludes the chapter.

2.1 Heterogeneous Architectures

Heterogeneity in computer architectures refers to computer systems that contain different types of processors. Each processor might have its own instruction set and memory and it is normally specialised for one particular task. As an example, the CPU and the GPU work together to compute and visualise information. Heterogeneous programming refers to the development of applications for heterogeneous architectures, such as CPUs and GPUs to solve a task. This thesis is focused on heterogeneous programming and, particularly, on GPU programming.

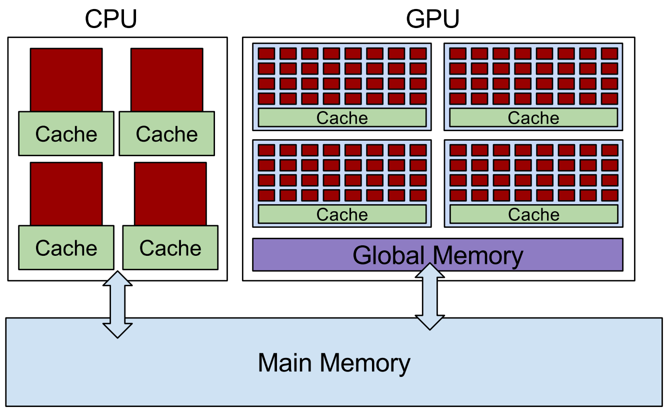


Figure 2.1: Simplified view of a heterogeneous computer system composed of a CPU and a GPU.

Figure 2.1 shows a simplified view of a heterogeneous computer system with a CPU and a dedicated GPU. Dedicated GPUs have access to the system main memory but the data has to be allocated and copied to the internal GPU global memory first, as Section 2.2.4 will show.

The GPU architecture is radically different from the CPU architecture. Figure 2.1 shows a CPU with four cores. Each CPU core has its own cache. Modern CPUs contain up to three levels of cache. For simplification, this figure only shows one level of cache. Alternately, GPUs provide multiple cores organised in groups of cores. Each group of cores contains its own cache. Dedicated GPUs have their own global memory, as the one shown in Figure 2.1. To allow the communication between the CPU and the GPU, first the CPU has to send data to the GPU global memory and then execute the corresponding application.

Data Parallelism Applications written for GPUs are good at exploiting **data parallelism**. This corresponds to applications where multiple data elements compute the same instruction in parallel in the multiple computing units of the GPU. More precisely, GPU architecture follows the model Single Instruction Multiple Data (**SIMD**) or its variant for GPUs called Single Instruction Multiple Thread (**SIMT**). This means that one single instruction is executed multiple times by many threads using different elements from the input data set. As Section 2.3 will describe, this type of parallelism can be easily exploited using parallel skeletons.

GPUs also support other types of parallelism, such as task parallelism, in which different tasks can be executed at the same time on the GPU. This thesis is focused on exploiting data parallel applications for interpreted and high-level programming languages on GPUs.

GPU Program Execution The CPU issues compute kernels for execution on the GPU. Then, a GPU scheduler organises those threads in blocks and sends them to compute units (CU) of the GPU. GPUs normally contain around 12-16 compute units, called also stream multiprocessors. Each CU contains schedulers that map each instruction to a bunch of physical GPU cores. Most GPUs issue groups of 32 threads together (called *warps*). Threads within the same *warp* execute the same instruction but using different data items from the input data set.

General Purpose GPU (GPGPU) Programming

GPUs are hardware dedicated to visualisation and rendering. GPUs were originally designed to process information of hundreds or thousands of pixels in parallel. One of the main features is that GPUs are very good at hiding latencies and maximising throughput and exploiting parallelism. In addition, they consume less energy and provide computation power equivalent to hundreds of equivalent CPUs.

For all of these reasons, GPUs have become an attractive hardware for researchers, data centres and industry companies to program general purpose applications. This means that GPUs can be used not only for visualisation and rendering, but also for general purpose computing, and specially, for data parallel applications.

There are a few programming languages and standards that people use to program GPUs. CUDA and OpenCL are the two most common and popular ones. These programming models and languages allow executing C applications on GPUs.

CUDA (Compute Unified Device Architecture) [CUDA 07] is an application programming interface and a programming model created by NVIDIA in 2007 for executing a program written in C into the GPU. A CUDA program can only be executed on NVIDIA GPUs. OpenCL (Open Computing Language) [OpenCL 09] is a standard created by the Khronos Group in 2008 that can execute parallel applications on CPUs, GPUs, FPGAs or any other kind of device that supports OpenCL. Although there are differences between CUDA and OpenCL, they share the parallel programming model. In general, any OpenCL program can be converted into a CUDA program. Next section reviews OpenCL and explains the main parts of the OpenCL specification.

2.2 OpenCL

OpenCL [OpenCL 09] (Open Computing Language) is a standard for heterogeneous programming on multiple devices. OpenCL [OpenCL 09] was initially proposed by Apple in collaboration with Intel, AMD, NVIDIA and Qualcomm. The initial OpenCL proposal was presented to the Khronos Group and it was finally released in 2008. The OpenCL standard is defined in four parts:

1. Platform model: it specifies the **host** and coordinates the OpenCL **devices** (e.g. GPUs).
2. Execution model: it defines how the host and OpenCL devices are coordinated with each other. It also defines OpenCL kernels (functions that are executed in OpenCL devices) and the program that is executed on the host.
3. Programming model: it defines the data and the task parallel programming models in OpenCL as well as how synchronisation is performed between tasks.
4. Memory model: it defines the OpenCL memory object types as well as it provides an abstraction for the memory hierarchy of OpenCL devices.

The rest of this section reviews the most important concepts of each part of the OpenCL architecture.

2.2.1 Platform Model

Figure 2.2 shows the two main components of the OpenCL platform. An OpenCL platform consists of one host connected to one or more OpenCL devices. An example is a CPU as a host, and then multiple GPUs connected to the CPU working as OpenCL devices. A CPU can also work as another OpenCL device.

Each device is composed of multiple compute units (CU) and each compute unit is composed of multiple processing elements which execute the final instructions. To make an analogy, a device is similar to a CPU. The CPU is composed of multiple cores, so, each core is a compute unit (CU). In its turn, each core contains multiple units to execute SIMD (Single Instruction Multiple Data) instructions. These units are the equivalent of the processing elements in each CPU core.

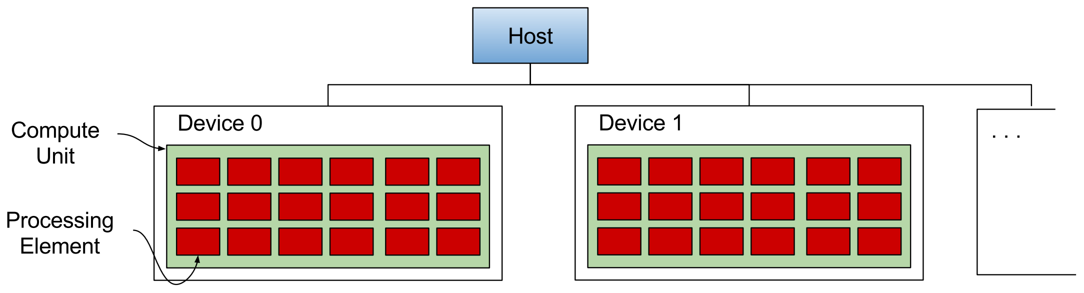


Figure 2.2: OpenCL Platform

```c
void vectorAddJava(int n, const float *a, const float *b, float *c) {
    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }
}
```

Listing 2.1: Vector addition implemented in C

2.2.2 Execution Model

The execution model is composed of two parts: the kernel and the host program. An OpenCL kernel is a function that will be executed in an OpenCL device. Kernels are implemented in a C-style using a subset of C99 standard language with extensions to specify OpenCL qualifiers and types. The host program defines all the data structures that are needed to execute an OpenCL program on an OpenCL device.

Kernels Listing 2.1 shows a function written in C that performs a vector addition. This is a very simple application where a `for` loop is used to index the elements of the input arrays and compute the corresponding values.

Parallelisation in OpenCL is implicit by mapping an input application into N-dimensional index space. Listing 2.2 shows the OpenCL implementation for the vector addition. There are a few differences. First of all, there is no loop in the OpenCL code. The index is obtained by invoking the OpenCL function `get_globalid(0)` which returns the thread-id that will execute this code. Each OpenCL thread-id corresponds to a **work-item** in the OpenCL terminology. Other differences are the new qualifiers `kernel` token, which indicates the function is an entry point for OpenCL kernel, and `global`, which defines the passed variables in the global memory on the OpenCL device.

```
kernel vectorAddOpenCL(global float *a, global float *b, global float *c) {
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

Listing 2.2: Kernel of vector addition application implemented in OpenCL

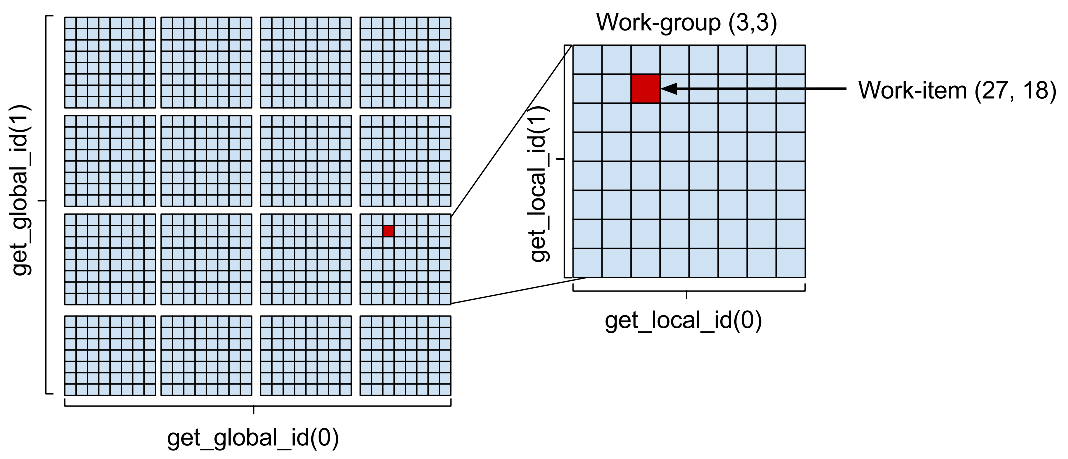


Figure 2.3: Example of relationship between OpenCL work-groups and work-items.

The host program enqueues an OpenCL kernel in a queue of commands. When the kernel is executed, an index space for running the OpenCL kernel on the device is defined. A kernel can be indexed in 1D, 2D or 3D. Figure 2.3 shows an example of a 2D index space that helps to understand how an OpenCL kernel is indexed. Work-items execute exactly the same function using different data items from the input data set. The right side of the Figure 2.3 highlights a work-item inside a block. Work-items can be grouped together to form a work-group. All the work-items in the same work-group are executed on the same device. These work-items in the same group can be synchronised and share memory. Different work-groups can not be synchronised between them.

Figure 2.3 shows 16 blocks. Each block is called **work-group** in OpenCL and can be identified by the block-id in the x and y dimensions. For example, the highlighted block corresponds to the block (3, 3). Each block contains 64 work-items organised in a block of 8*x*8 elements.

Work-items and work-groups can be accessed by the kernel using OpenCL reserved functions. For example, as shown in Listing 2.2, the function `get_global_id(0)` is invoked to index the x-dimension space inside the OpenCL kernel.

Host programs The part that manages and orchestrates the OpenCL execution is the host program. The host part is the responsible for the following tasks:

1. Query the OpenCL platforms that are available.
2. Query the OpenCL devices that are available within a platform.
3. Create a set of device objects (one per device).
4. Create an OpenCL context that is used to push commands to be executed on a OpenCL device.
5. Create an OpenCL command queue.
6. Compile and create OpenCL kernel objects.
7. Create the OpenCL buffers and copy the data from the host to OpenCL devices.
8. Invoke the OpenCL kernel with the corresponding parameters.
9. Send the result from OpenCL devices to the host via the OpenCL buffers and command queue.

OpenCL provides an Application Programming Interface (API) that allows programmers to perform all of these operations.

2.2.3 Programming Model

This section explains how an OpenCL program is executed on an accelerator or a GPU and how an OpenCL application is mapped to the architecture. Part of the programmability is defined in the OpenCL execution model explained in Section 2.2.2.

The host program enqueues a kernel to be executed on an specific OpenCL device. When the OpenCL kernel is launched, it is executed following the SIMT (Single Instruction Multiple Thread) parallel programming model where the same instruction is executed multiple times with different data items from the input data set). Each input element maps to a thread on the OpenCL device. The GPU groups threads in chunks of 32 or 64 threads (depending on the target GPU architecture) and distribute them to each compute unit. Compute units, as explained in Section 2.1, have schedulers that issue one instruction per cycle and assign a group of physical cores to execute that instruction with different input data.

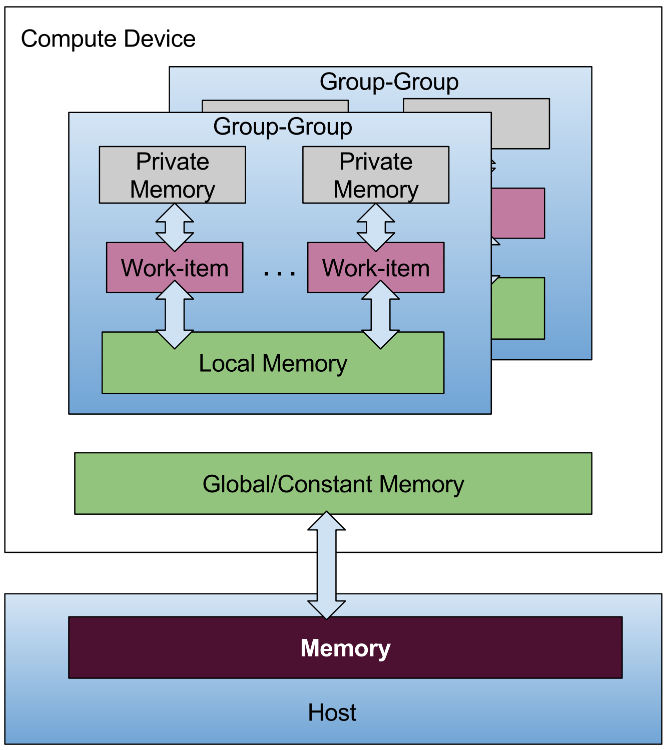


Figure 2.4: OpenCL device and host memory hierarchies.

OpenCL is a low-level programming model where hardware abstractions are exposed to programmers. Therefore, OpenCL is made for experts programmers, not only with parallel programming knowledge, but also with hardware architecture knowledge.

2.2.4 Memory Model

The host and devices have their own memory systems, and in some cases, they are situated in separated locations (e.g. dedicated GPUs). Figure 2.4 shows the OpenCL memory hierarchy and the interconnection between the host and the compute device.

The compute device has a global memory, which is shared across all the work-groups. There is also a region dedicated to constant memory, which is in global memory but read-only, allowing to increase the speed of memory accesses. Then, each work-group has its own local memory. All the threads within the same work-group can access the local memory. Moreover, each work-item has a set of registers available in private memory. All levels of memory, except for private memory, are exposed to programmers. The capacity of all the different memory hierarchies can be queried by using the OpenCL API.

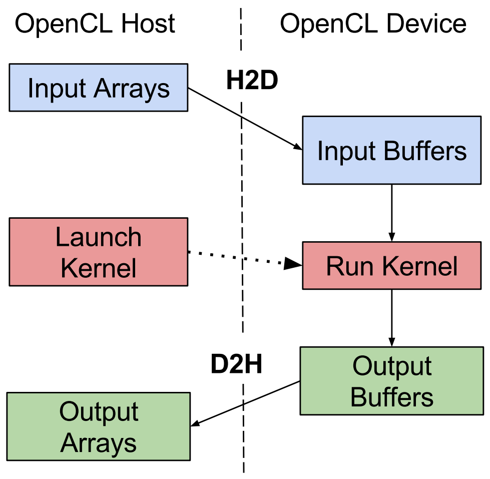


Figure 2.5: OpenCL work-flow between the OpenCL host and an OpenCL device.

Data Transfers Figure 2.5 shows the typical OpenCL work-flow between the OpenCL host and an OpenCL device. The host allocates the memory for the input arrays. Then, the host needs to allocate the corresponding input buffers on the OpenCL device and performs the data transfer (copy from the host to the device, *H2D*). After that, the host launches the kernel and this is executed on the OpenCL device. The result is then transferred to the host through an operation from the device to the host (*D2H*).

2.3 Structured Parallel Patterns

Parallel computing is a way of exploiting parallel resources of the hardware to speed-up applications by dividing tasks in smaller and more manageable ones across many processing units. These processing units can be located on the same machine, using shared memory, or on different machines that are interconnected by a network.

In order to optimise, programmers must know low-level software and hardware details. The previous section Section 2.2 just shown how the OpenCL programming model is used for GPU programming or any other type of OpenCL-supported accelerator (e.g., CPUs and FPGAs). Developers can use GPUs by implementing applications in OpenCL. However, if other types of architectures are involved (not only GPUs but for instance, a multi-core system with multiple CPUs), the OpenCL programming can be combined with shared memory programming models, such as OpenMP, to obtain better performance. Consequently, applications have to be rewritten with the underlying technology in mind. Therefore, applications are implemented by mixing low-level details for optimising and parallelising with the algorithms themselves. From the software engineering perspective, this way of programming makes it difficult to adapt the code to new hardware and to maintain it.

A smarter way of programming could be the separation of algorithm from implementation and creating patterns in which applications can be easily ported and parallelised. Murray Cole introduced the concept of algorithmic skeletons [Cole 91]. Cole proposed algorithmic skeletons to solve recurrent problems in parallel and distributed computing. They are, in fact, widely used nowadays in parallel and heterogeneous programming. Algorithmic skeletons are presented by Cole as an alternative to implement parallel applications focusing on the task itself (the algorithm) rather than the technology. In this way, programs are not only much easier to parallelise, but also to maintain and modify the implementation according to new parallel programming models and parallel hardware. For the rest of this document, the terms algorithmic skeletons and parallel patterns are interchangeable to refer to the same concept.

One part of this thesis exploits the concept of algorithmic skeletons for heterogeneous programming for high-level programming languages. Although there is a wide variety of algorithmic skeletons for parallel programming based on data, task and pipeline parallelism, this thesis covers the two main parallel algorithmic skeletons; **map** and **reduce**. Other algorithmic skeletons can be built based on these two. There are many others, such as **scan**, **stencil**, **gather**, **scatter** and **fork-join**, that are not covered in this thesis. The rest of this section explains map and reduce computation.

Map computation One of the most basic parallel patterns is the **map** parallel skeleton which receives a user defined function (UDF) and a set of input arrays. The map function computes the user function for each element of the input data set. Map computation corresponds to embarrassingly parallel applications where each element of the output data set is computed independently from the rest. Therefore, each item can be computed separately without any data dependency. This means that, for example, to compute the element i, the map computation only needs the input element in the position i and computes the output in the same position. There are a lot of programming languages that support it such as Python, Perl, Ruby, Lime, Haskell as a primitive or built-in in the language.

Reduction In combination with the map parallel skeleton, there is also another important skeleton, the **reduction**, which basically computes an operation with an array input and reduces it into a scalar value. Map/reduce parallel skeletons are the most basic and powerful patterns. Many applications can be expressed as a combination of these two as this thesis will show. Also, these are the basic primitives for other related frameworks, such as Apache Spark [Zaha 10], Apache Flink [Carb 15], Hadoop [Mani 14] or Google Map/Reduce [Dean 04].

2.4 Managed Languages and Runtimes

This thesis is focused on managed runtime programming languages as a software architecture, and, specially, on the Java and Java Virtual Machine (JVM). This section explains the main concepts related to the JVM and how it works. It also presents Graal, an aggressive JIT compiler framework for the JVM, and Truffle, a DSL for implementing programming languages on top of Graal and the JVM. This thesis extends Truffle and Graal to execute R and Java programs on GPUs.

2.4.1 Java

Java [Gosl 95] is a general-purpose programming language. The main characteristic of Java is that it is an object-oriented programming (OOP) and platform independent language. Java is, at the time of writing this thesis, the most popular programming language [TIOB 17] and is present everywhere. Java is used in desktop systems, mobiles, e-readers and internet.

Java is a simple, portable across architectures, multithreaded, dynamic, distributed, object oriented, high performance, robust and secure programming language [Gosl 95]. It follows the philosophy ¨*write-once-run-everywhere*¨ where the input source code is compiled once to an intermediate representation that is common for all Java platforms and executes it in multiple platforms without any modification in the source code.

Java is widely used in industry and academia. One of the most common cases is used for programming user applications for the Android operating system. Java is composed of two parts, the Java language and the Java platform [Gall 14]. The Java Platform contains a high-level API (Application Programming Interface) exposed to the user and the Java Virtual Machine (JVM). The JVM is the software machinery that is needed to execute Java programs on a specific hardware and operating system.

```
1 // Hello world application in Java
2 public class JavaExample {
3         public static void main(String[] args) {
4                 System.out.println("Hello world");
5         }
6 }
```

Listing 2.3: Hello World application implemented in Java

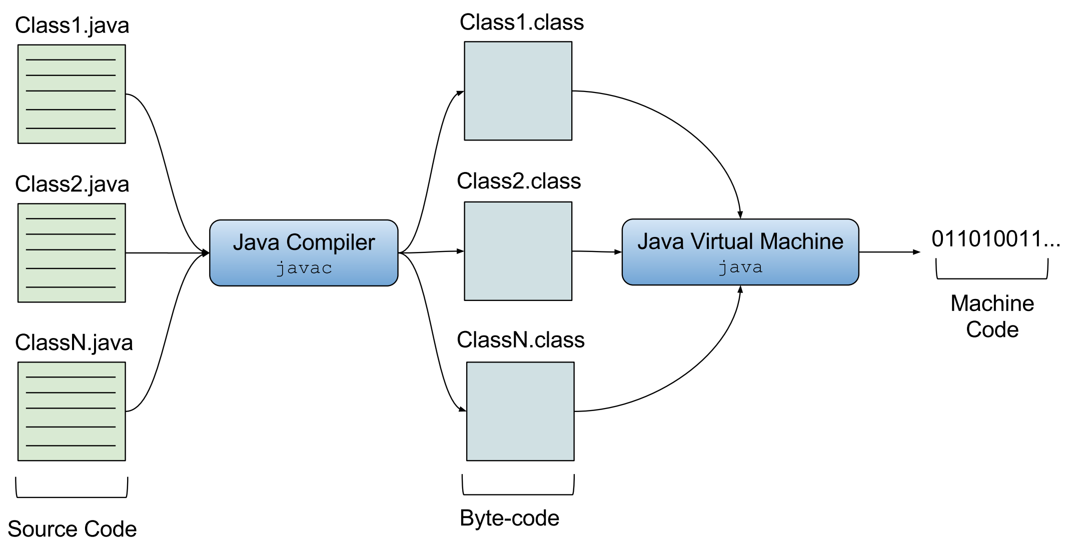


Figure 2.6: Java compiler process

Listing 2.3 shows the *hello world* application implemented in Java. This program only prints the string `"hello World"` on the screen. Line 2 creates a class called `JavaExample`. Line 3 defines a Java method (`main`) which tells the VM that this is the first line to execute.

2.4.2 Java Virtual Machines

Java programmers provide the source code in a set of Java classes. These classes are compiled to an intermediate representation using the Java compiler (`javac`) as shown on the left side of Figure 2.6. This intermediate representation is called **Java bytecode** and it is hardware agnostic. This means that once the Java compiler generates the Java bytecode, this bytecode can be executed on any machine provided with a Java platform. Then, the JVM loads, verifies and executes Java bytecode in a specific machine.

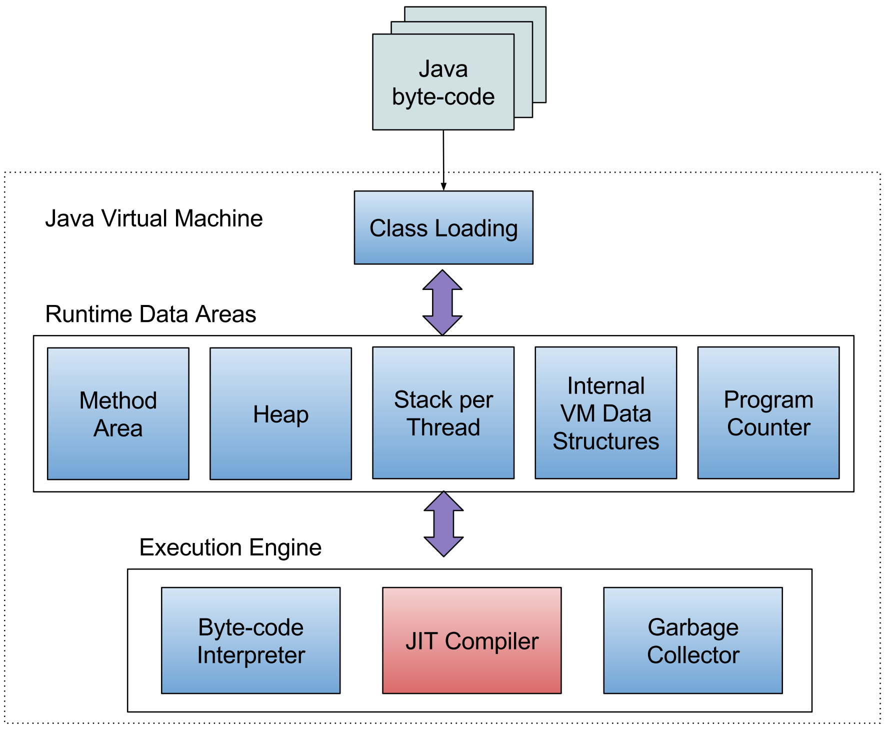


Figure 2.7: Main components of the Java Virtual Machine (JVM)

JVM main components

The JVM contains a set of software components to execute the Java bytecode in the target hardware. Figure 2.7 shows an overview of the main components of the Java Virtual Machine. When a Java application is executed, the JVM first loads and verifies the input bytecode. Then, it loads the input classes in the runtime data area. For example, methods and constants are loaded into the method area and constant pool of a reserved area of the runtime. New Java objects are allocated on a heap area.

The JVM starts running the program in a bytecode interpreter. The bytecode interpreter reads one bytecode after another and translates it, at runtime, into machine code. This translation process is very slow. In order to accelerate the program execution, the JVM contains a key component to achieve good performance, the **JIT compiler**. The JIT compiler is represented in red to indicate that it is one of the key components used in this thesis for GPU optimisations and code generation. When part of the Java program is executed multiple times, the VM optimises and compiles a whole Java method or a whole loop at runtime into efficient machine code. If the method or the loop is executed again, the JVM switches from the interpreter code to the efficient machine code.

The JVM uses a heap in the runtime data area to allocate Java objects. Those objects are used by the execution engine to allocate the required memory. In Java, objects are automatically de-allocated by a **garbage collector** [McCa 60]. The garbage collector, originally designed for the Lisp programming language in 1960, is a program that organises the heap by detecting dereferenced Java objects, compacting contiguous space to allocate new objects and removing the dereferenced ones automatically. Therefore, programmers do not have to manage the memory at all.

This thesis is focused on GPU JIT compilation and runtime execution for managed programming languages to speed-up Java and R programs. The next section presents the main concepts related to JIT compilation and runtime management and it explains how the JIT compiler works.

Java HotSpot Virtual Machine

Originally implemented by Sun Microsystems in 1999, Java HotSpot is an optimising compiler for the Java Virtual Machine which includes two interpreters and two JIT compilers. The two interpreters are a template interpreter and a C++ interpreter. The template interpreter offers better performance at the cost of facility to port to other computer architectures. The bytecode interpreter is built at runtime according to the input Java bytecode of the program. This interpreter has a table to map the conversion between the input bytecode and the memory region in which the set of CPU instructions are stored. The two JIT compilers available in Java HotSpot VM became generally available in Sun JVM 1.3. The rest of this section introduces these two JIT compilers.

How the JVM Executes Bytecode Figure 2.8 shows a representation of the interpreter execution (one interpreter for simplification) and the two compilers in Java HotSpot VM. After Java classes are loaded by the class loader, the VM starts executing each bytecode from the input Java program in the bytecode interpreter, and, at the same time, it profiles the program in order to be able to apply better optimisations in later phases. If the input program is executed frequently, the profiler detects the ¨hot spot¨ parts of the program and marks them as candidates for compilation. The compilation from the Java bytecode into an efficient machine code in the Java HotSpot VM happens at runtime (just-in-time compilation). In the Java HotSpot VM there are two available JIT compilers used for different levels of tier compilation: the client compiler [Kotz 08], also called C1 compiler, is used in a high-tier compilation for fast startup, and the server compiler [Pale 01], also called C2 compiler, is used for a low-

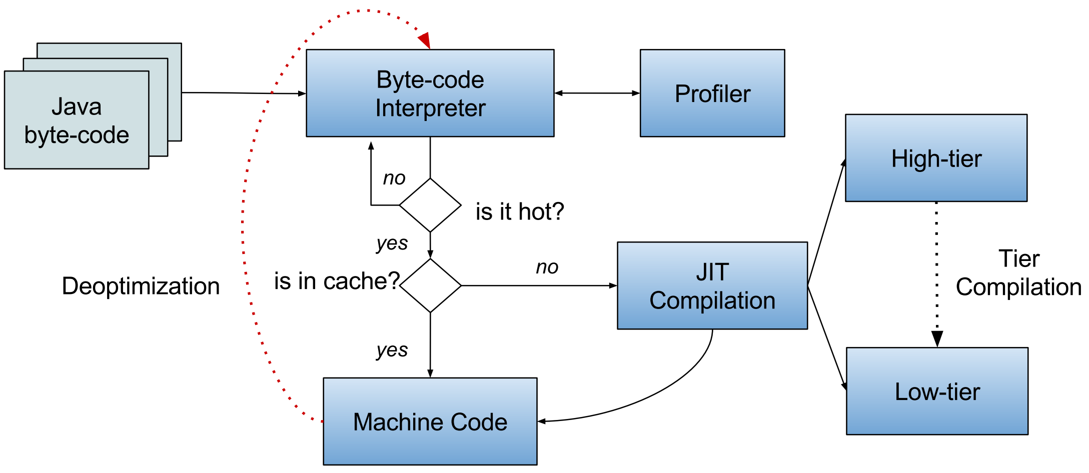


Figure 2.8: Java JIT compilers in HotSpot VM

level tier compilation which applies a set of aggressive optimisations. The difference between these two compilers is the type of optimisations and the moment in which they are applied based on profiling information.

Tier Compilation There is also a third option to compile the code, called tier compilation, in which the JIT compiler starts to compile as fast as the low-tier compiler and, as the code gets hotter, it continues compiling to the low-tier as shown in Figure 2.8. This mode of compilation is the default JIT compiler from Java 8.

Deoptimisations The JIT compiler use speculations to compile the code. Speculations are optimistic optimisations that are taken to achieve better performance that are highly optimised for the common cases. However, if a speculation is assumed wrongly, the virtual machine needs to catch the missing speculation, it invalidates the compiled code and falls back to the interpreter. This process is called deoptimisation [Holz 92]. The JIT compiler inserts guards to capture the miss-speculation and falls back to the bytecode interpreter.

2.4.3 Graal

Graal [Dubo 13a, Dubo 13b, Simo 15, Grim 13, Dubo 14] is an aggressive JIT compiler for the Java platform implemented in Java. It replaces the JIT compilers in Java HotSpot VM. Moreover, Graal can be invoked under demand to compile specific methods to machine code at runtime. Graal parses the input bytecode and builds another intermediate representation called **Graal IR**. Graal is now part of the OpenJDK

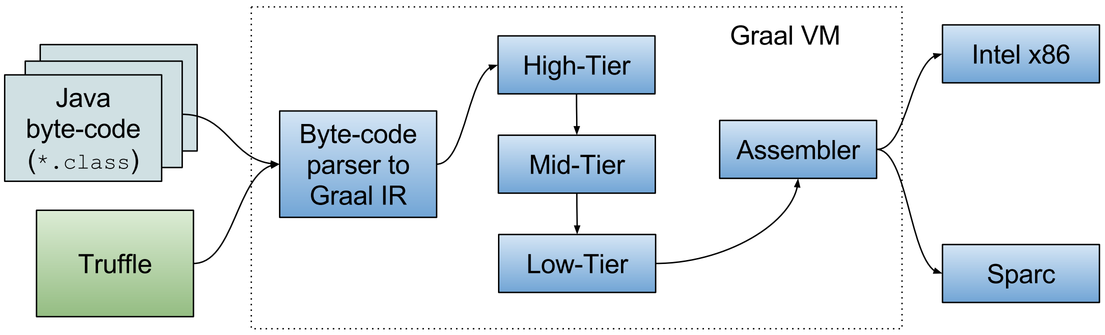


Figure 2.9: Graal Compiler Workflow

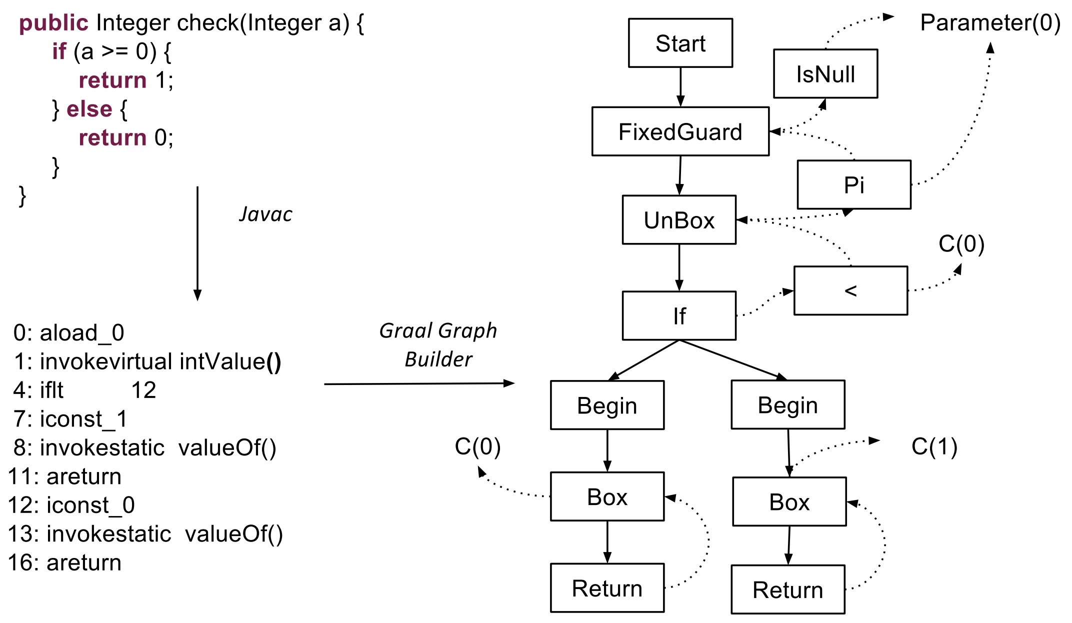
project [Open 17]. Graal reuses the rest of the components available in Java HotSpot VM such as the garbage collector, class loader and the bytecode interpreter.

Graal Compiler Workflow Graal compiles Java bytecode to efficient machine code at runtime. Figure 2.9 shows the Graal's compiler workflow. Graal first parses the Java bytecode and then it builds a graph representing control flow and data flow. This graph represents the Graal intermediate representation (Graal IR). Graal uses this graph to apply compiler optimisations and generate efficient machine code.

Graal has three different levels of tier compilation: high-tier, mid-tier and low-tier. High-tier applies architecture independent optimisations such as inlining, constant propagation, value global numbering, partial escape analysis [Stad 14b]. In the mid-tier is where Graal applies specific memory optimisations and in the low-tier, Graal applies machine specific optimisations and it prepares the graph for lowering the CPU machine code. In the last phase of the compiler, Graal applies register allocations and peephole optimisations before emitting the target machine code [Stad 14a].

Graal IR The Graal intermediate representation (Graal IR) [Dubo 13b] is a directed acyclic graph based on SSA [Cytr 91] (static single assignment) form, that is, each variable in the IR has a unique identifier. The SSA representation facilitates the compiler to apply optimisations such as constant folding, constant propagation, etc.

The graph represents both control flow and data flow. Control flow nodes are connected by their successors. Data flow nodes are connected by their input value dependencies. Nodes can be classified in two big categories: fixed nodes and floating nodes. The idea behind this classification is that floating nodes can be moved around the control flow graph as long as it preserves the program semantics. Fixed nodes correspond to control flow nodes, such as `IfNode` or `ForNode`.



```
public Integer check(Integer a) {
    if (a >= 0) {
       return 1;
    } else {
       return 0;
    }
}
```

```
 0: aload_0
 1: invokevirtual intValue()
 4: iflt        12
 7: iconst_1
 8: invokestatic  valueOf()
11: areturn
12: iconst_0
13: invokestatic  valueOf()
16: areturn
```

Figure 2.10: Example of a Java method at the top left with its corresponding Java bytecode bottom left and the Graal IR on the right.

Figure 2.10 shows an example of a Java method that performs a check of an input value. The method returns 0 if the input value is negative, otherwise it returns 1. Note that the input and output are Integer objects. The Java method is shown on the top left of the Figure. The bottom left part corresponds to the Java bytecode of this method and the right part shows the control flow graph (CFG) in the Graal IR form. Each Graal graph has a **start** node. Black arrow lines represent control flow and dash lines represent data flow.

All CFG in Graal begins with a start node as shown on the right side of the figure. The CFG receives a parameter (the input parameter `a`) and checks if this value is **`null`**. Graal inserts a new node, called `IsNullNode` to perform this check. If the value is **`null`**, it deoptimises to the bytecode interpreter. This is the node `FixedGuardNode` in Figure 2.10. If this value is not **`null`**, then it **unboxes** the value (from Integer class to **`int`**) and checks if it is less than zero and finally it performs the corresponding operation.

2.4.4 Truffle

Truffle [Wurt 13] is a framework for implementing AST interpreters on top of a Java Virtual Machine (JVM). The Truffle API contains various building blocks for a language's runtime environment and provides infrastructure for managing executable code, mainly in the form of Abstract Syntax Trees (ASTs). The Truffle framework was originally designed for dynamic typed programming languages such as JavaScript, R or Ruby. However, there is no restriction to execute static typed languages, such as C and Pascal.

AST interpreters are a simple and straightforward technique to build an execution engine for a programming language. The source code is transformed into a tree of nodes, and the `execute` method of each node defines its behaviour.

Specialisation in Truffle For dynamic programming languages, even seemingly simple operations, such as adding two values, can perform a multitude of different tasks depending on the input value types and the overall state of the runtime system. The Truffle AST nodes start out in an uninitialised state and replace themselves with specialised versions geared towards the specific input data that are encountered during execution. As new situations are encountered, the AST nodes are progressively made more generic to incorporate the handling of more inputs in their specialised behaviour. This is called *specialisation*. At any point in time, the AST encodes the minimal amount of functionality needed to run the program with the inputs encountered so far. Most applications quickly reach a stable state where no new input types are discovered.

Writing nodes that specialise themselves involves a large amount of boilerplate code that is tedious to write and hard to get right under all circumstances. The Truffle DSL provides a small but powerful set of declarative annotations used to generate this code automatically.

Partial Evaluation Partial evaluation [Futa 99] is a technique for program optimisations by specialisation. This concept was originally presented by Futamura, who creates three types of specialisations called the three *Futamura projections*. The first Futamura projection compiles the interpreted that has been specialised. By repeating this specialisation process, the other two Futamura projections can be derived. The second Futamura projection yields a compiler by applying the first projection and the third Futamura projection yields a tool to generate compilers. Truffle framework is based on the first Futamura projection [Wurt 13, Wurt 17], where the interpreter is specialised and then compiled to machine code.

Efficient JIT Compilation with Graal using Partial Evaluation The specialisation of AST nodes together with the Truffle DSL allow the interpreter to run efficiently, e.g., to avoid boxing primitive values in certain situations. However, the inherent overhead of an interpreter which dispatches `execute` calls to the AST nodes cannot be removed.

To address this, Truffle employs Graal for generating optimised machine code. When Truffle detects that the number of times an AST was executed exceeds a certain threshold, it will submit the AST to Graal for compilation. From that point, Graal optimises the graph as any other Java program, because the AST interpreted is implemented in Java. Graal compiles the AST using partial evaluation, which essentially inlines all execute methods into one compilation unit and incorporates the current specialised AST to generate a piece of native code that works for all data types encountered so far. If new data types are encountered, the compiled code will deoptimise and control will be transferred back to the interpreter which modifies the AST to accommodate the new data types. The AST is then recompiled with the additional functionality.

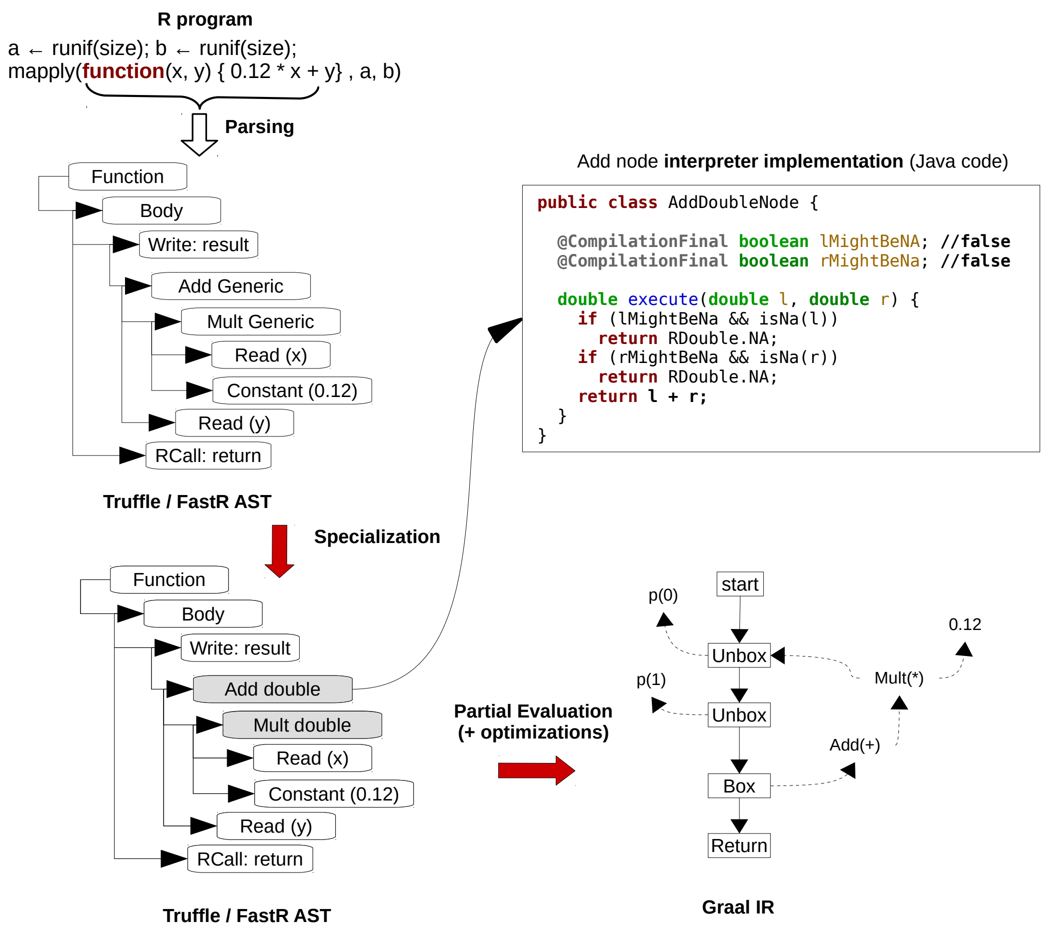
Truffle and Graal workflow: example in *FastR* To illustrate how Truffle and Graal work together to obtain maximum performance from a dynamic programming language, the rest of this section shows an example written in R and traces how Truffle and Graal compile the input code to efficient machine code.

Figure 2.11 shows a program executed using *FastR*, an implementation of the R language built using Truffle and Graal. The program shown in the upper left corner executes a `function` for each pair of elements from vectors `a` and `b` using the `mapply` function. On execution, FastR creates the AST of the function passed to `mapply`, as shown on the left side of the figure. This AST is generic and not yet specialised to the input data used in the execution.

As it is executed, the AST rewrites itself to the state that is shown at the bottom left of Figure 2.11. The addition and multiplication nodes are specialised for the `double` data used in the example. The Java code in the upper-right of the Figure shows an example of implementation for the `AddDoubleNode` AST node. The `AddDoubleNode` is the variant of the addition node specialised for `double` values. FastR calls the `execute` methods of the AST nodes inside the interpreter.

In order to respect the semantics of the R language, the `AddDoubleNode` needs to handle `NA` (not available) values. `NA` is a special marker that represents the absence of a value, e.g., a missing value inside a collection. The `execute` method of `AddDoubleNode` handles `NA`s by returning `RDdouble.NA` when one of the two operands is `NA`.

As an optimization, the infrastructure that performs arithmetic operations on vec-



```java
public class AddDoubleNode {

  @CompilationFinal boolean lMightBeNA; //false
  @CompilationFinal boolean rMightBeNa; //false

  double execute(double l, double r) {
    if (lMightBeNa && isNa(l))
      return RDouble.NA;
    if (rMightBeNa && isNa(r))
      return RDouble.NA;
    return l + r;
  }
}
```

Figure 2.11: Execution of an R program in FastR via Truffle and Graal.

tors sets the `lMightBeNA` and `rMightBeNA` fields to `true` only if a vector that may contain `NA`s was encountered as an operand. As long as these flags remain `false`, no checks for `NA` values in the operands are necessary. Truffle uses *compiler directives* to convey this information to an optimising runtime. The `@CompilationFinal` directive is used in the example to indicate that the Boolean values can be assumed to be constant during compilation.

The partial evaluation performed by Graal transforms the FastR AST into the Graal IR shown on the lower right of Figure 2.11. Information specified in the compiler directives is used by the partial evaluation to perform optimisations. In the example, both branches visible in the node's `execute` method are removed based on the knowledge that the operands can never be `NA`. Therefore, partial evaluation optimises the generated code by removing logic from the interpreter which is only required for exceptional cases.

2.5 Summary

This chapter has presented the terminology and the main terms that are needed for understanding the rest of this thesis. This chapter has introduced the graphics processor unit (GPU) architecture and the OpenCL programming model. It has shown the basic steps to implement GPU applications using OpenCL. This chapter has also introduced the terminology of algorithmic skeletons and managed languages. It has focused on JIT compilation and all the techniques to compile, at runtime, the Java bytecode of an input application to efficient machine code. It has presented Truffle and Graal, the two compiler frameworks used in this thesis, for compiling Java and dynamic programming languages to efficient machine code on top of the JVM. It ends with the introduction of the compilation of high-level and dynamically typed languages, such as R, Ruby or JavaScript using the Truffle and Graal compiler frameworks.

Chapter 3

Related Work

This chapter presents the state-of-the art works related to this thesis. Section 3.1 discusses different design and implementation approaches for GPU programming. Section 3.2 presents the related works based on algorithmic skeletons. Section 3.3 presents another alternative to program heterogeneous systems using external libraries. The main contributions of this thesis are based on existing programming languages that are executed on top of the Java Virtual Machine (JVM) for GPU programming. Section 3.4 shows related works for using GPUs from the Java programming language. It analyses, in detail, different approaches of using Java and the JVM for programming GPUs. Section 3.5 presents different language implementations on top of Truffle and Graal. Section 3.6 presents related works on GPUs programming from dynamic and interpreted programming languages. Finally, Section 3.7 concludes and summarises this chapter.

3.1 Languages for Heterogeneous Programming

This section presents different programming language approaches for GPU programming, starting from directive-based programming models to the design of explicit parallel programming languages. Section 3.1.1 presents directive-based programming models and Section 3.1.2 discusses the explicit parallel programming languages. Section 3.1.3 presents the most common intermediate representations for heterogeneous computing.

3.1.1 Directive-based Heterogeneous Programming

Directive-based programming allow programmers to extend easily sequential applications with annotations in the source code to indicate where the parallelism is located. This information is used by an optimising compiler to generate the corresponding parallel and heterogeneous code. The most well-known case is OpenMP [Dagu 98], a parallel programming standard for multi-core programming, where developers annotate loops with OpenMP *pragmas*. These *pragmas* are not part of the language grammar. It is part of the preprocessors' work to transform these *pragmas* into valid C/C++ or Fortran code.

Low-level programming languages Some approaches have followed the successful case of OpenMP for heterogeneous programming for languages like C/C++ and Fortran. Lee et al [Lee 09] presents a source-to-source compiler that translates C programs with OpenMP directives into a valid C and CUDA programs to accelerate on GPUs.

OmpSs [Elan 14] is a programming model for heterogeneous architectures that extends the set of directives available in OpenMP using OpenCL.

The *hiCUDA* [Han 11] compiler is also a directive based programming model for executing a C/C++ application on GPUs through CUDA. As shown in Section 2.2, an OpenCL program first allocates memory on the device, performs the data transfer and then it executes the OpenCL kernel on the heterogeneous device. CUDA and OpenCL are very similar programming models and they share the same concepts of how applications are programmed and executed on GPUs. *hiCUDA* provides a higher-level abstraction in the form of *pragmas* that maps almost one-to-one with CUDA operations. When using *hiCUDA*, developers specify where variables are allocated (e.g. GPU global memory), which variables are copied in and which variables are copied out, the number of threads to be launched by the CUDA kernel and how they are distributed on the GPU. This high-tuning of the application using *hiCUDA pragma* clearly influences the performance, where low-level details about the architecture are exposed to programmers. This approach, even if using *pragmas* still remains low-level for non-expert GPU programmers.

Following pragma directives extensions for heterogeneous computing, the PGI Accelerator Model [Wolf 10] is another higher-level programming model for GPUs. Developers annotate loops in the sequential C code and the compiler statically generates the corresponding CUDA kernel. Compared to *hiCUDA*, in PGI Accelerator only a few annotations are needed. PGI is, in fact, a very conservative compiler where if a

CUDA kernel can not be generated, it falls back and executes the original sequential version.

CAPS HMPP [Bodi 09] is a similar programming model where the code is grouped in so called *codelets*. Codelets are user functions in the input program annotated with HMPP directives. They correspond to regions to parallelise and execute on GPUs using CUDA or OpenCL. HMPP has some restrictions, such as it only compiles so called pure-functions, functions that always produce the same result and do not have side effects, among other restrictions. Then, loops are explicitly annotated with HMPP directives. In the annotations, the target architecture has to be specified (for example CUDA, or OpenCL). This approach is a slightly higher approach than *hiCUDA* but it is a much lower-level approach compared to PGI Accelerator Model.

The OpenACC standard was created at the Supercomputing Conference in 2011. This standard allows developers to program heterogeneous devices with directives following the OpenMP and PGI approaches. Only a few annotations are required to specify that a loop has to be parallelised and executed on a GPU. It depends on the implementation, the technology that is used underneath. For instance, *accULL* [Reye 12a, Reye 12c, Reye 12b, Gril 13] is an OpenACC implementation that generates CUDA and OpenCL kernels statically (at GCC compilation time).

Ravi *et al.* [Ravi 13] proposed a new set of directives to add relaxed semantics to the input program. This means that these directives suggest the compiler that a section of the program can be parallel, and the runtime decides if finally the annotated code will be executed on the accelerator. Ravi's work focuses on Intel Xeon Phi accelerators. The goal with new directives is to create a model where parts of the problem can be executed into the CPU and the other part into the GPU simultaneously.

Based on the successful case of OpenACC, a new version of OpenMP was released in 2013, OpenMP 4.0, which includes support for accelerators and SIMD parallelism. The important difference between OpenACC and OpenMP is that OpenACC is focused only on accelerators, whereas OpenMP covers more general parallelism, such as task parallelism. OpenMP is also more verbose when accelerators are used, whereas OpenACC is much simpler to program.

High-level programming languages The explained related projects are focused on low-level programming languages like C/C++ and Fortran. This subsection presents the related projects for dynamic and interpreted programming languages, such as Python and Javascript.

```python
1 @cu
2 def saxpy(a, b, alpha):
3     return map((lambda x, y: alpha * x + y), a, b)
```

Listing 3.1: Saxpy example with Copperhead

Copperhead [Cata 11] is a high-level data parallel embedded language for Python. Copperhead defines a subset of Python 2.6 to compile to CUDA to execute on the GPU. It defines a set of Python decorators (annotations that are able to inject and modify code in Python functions) in the source code to compile to the GPU. Copperhead is a source-to-source compiler that translates, at runtime, a Python function into CUDA C code. It exploits parallelism through a set of primitives in the Python languages such as map and reduce. It targets nested parallelism by taking advantage of the GPU hardware parallelism features (e.g. by exploiting 2D and 3D CUDA kernels). Listing 3.1 shows an example of a Python program using Copperhead to execute on GPUs. Line 1 sets the annotation `@cu` (Python decorator) that enforces the following function to translate and compile to CUDA. The function contains a `map` (predefined primitive in Python) to express data parallel computation.

Numba [Lam 15] is a JIT optimising compiler for the Python programming language. Similarly to Copperhead, Numba uses annotations (in the form of Python decorators) in the source level for Python methods to give the compiler more information about how to compile the code. Numba has a special decorator to express that the annotated code should be used for generating CUDA code to target GPUs. Numba decorators for GPUs also include the data type of the input and output and CUDA thread selection. The Python code to be executed on the GPU is similar to the CUDA-C and OpenCL. Therefore, programmers must know the CUDA programming model.

Bytespresso [Chib 16] is a DSL on top of Java language for heterogeneous computing. It extends AST interpreters with annotations that are exposed to language implementers to facilitate the translation code from Java to CUDA to execute on GPUs.

Limitations of directive-based approaches Directive-based programming offers a nice to have feature for legacy code to use accelerator hardware such as GPUs, with minor modifications. However, programmability is not that easy. *hiCUDA* has directives to specify where the variables are allocated on the GPU memory hierarchy (e.g., shared or global memory). In OpenACC, there is also a directive to set up the number of threads and how they are distributed when the GPU kernel is launched (gans and workers in the OpenACC terminology). All of these parameters have a direct impact in performance. Ruyman *et al* [Reye 12a, Reye 13] studied the influence of these annotations for OpenACC implementations, such as PGI, HMPP and accULL (an open source OpenACC implementation), and depending on the values selected for thread distribution on the GPU, the performance varied among implementations.

Moreover some of the GPU frameworks specify the data type to be compiled to GPU code in the directives. That is the case of Numba. This is because Python is a dynamic typed language, where types can be changed at any point of the execution time. Ideally, programmers do not have to adapt the source code to execute their current applications on heterogeneous architectures.

3.1.2 Explicit Parallel Programming Languages

Another alternative to extend or use existing programming languages is to create new parallel programming languages. There are many works in this area.

Data-flow parallel programming languages StreamIT [Thie 02, Thie 10] is a programming language particularly designed for stream programming. Stream applications are commonly used in digital signal, audio and video processing. StreamIT programs have two main programming abstractions: filters (also called nodes) and communication channels. By using these two abstractions, programmers can provide a graph of dependencies to represent data and task parallelism. StreamIT applications can also follow the so called *hierarchical stream structure* which basically represents parallel patterns used to write stream applications. It contains the **pipeline**, **fork-join** (or splitter-joiner) and the **feedbackloop**.

Uduppa *et al.* [Udup 09] proposed an extension to the existing StreamIT that compiles to CUDA in order to execute on GPUs. It uses different profiling information to determine the best configuration of threads and blocks to execute on GPUs. Udappa's work also provides a novel buffer layout that exploits the memory bandwidth between the CPU and the GPU.

Lime Compiler [Duba 12, Baco 13] is an extension for the Java programming language that proposes a set of new operators to indicate to the compiler the regions to be offloaded to GPUs and FPGAs. Lime programming language contains the operator ¨=>¨, which provides tasks and pipeline parallelism. Lime statically generates OpenCL kernels for the tasks expressed with Lime heterogeneous operators. Lime also provides the map/reduce pattern as algorithmic skeletons through `@` and `!` symbols. When the compiler finds these new operators and symbols, it translates the input Lime code to OpenCL for executing on GPUs.

Similarly to StreamIT, Halide [Raga 13, Mull 16] is a promising functional programming language designed for graphics computation which uses pipeline and stencil computation. Halide is implemented as a Domain Specific Language (DSL) embedded in C++. Halide's main characteristic is that it separates the algorithm from the optimisations (called scheduling) and it makes easier to explore several optimisations. Still, in Halide optimisation exploration is performed by the programmer.

SYCL [SYCL 16, Reye 15] is a standard for heterogeneous programming built on top of OpenCL for C++ in a single source code. SYCL does not provide any language extensions, external libraries or separated kernel files. A SYCL compiler statically compiles a subset of the C++ program into the target platform (e.g. GPU). It makes use of SPIR to compile the GPU kernels and it uses OpenCL to manage and execute on CPU and GPUs.

Intel Cilk Plus [Robi 13] is a C++ based programming language that has been extended for with array notation for array parallelism. Intel Cilk Plus provides a C extension to support vector parallelism and task parallelism that allows map and reduce parallel operations. By using vector parallelism in Cilk Plus, programmers can take advantage of the low-level vector instructions available in Intel and AMD processors, such as SSE, AVX and AVX2 vector instructions sets [Krzi 16, Fume 13]. This allows programmers to compute multiple input elements in the same processor instruction. Intel Cilk Plus also contains new tokens to express task parallelism, such as `cilk_for` to parallelize for loops and `cilk_spawn` to launch new tasks in parallel.

Functional programming approaches NOVA [Coll 14] is a functional programming language designed for data parallelism on GPUs using the CUDA programming language. It contains a set of primitives that are used as algorithmic skeletons for data parallelism, such as map, reduce, scan, gather, or filter among others. An application written in Nova is compiled to an intermediate representation, NOVA IR, and then optimised. NOVA currently generates code for C and CUDA.

Futhark [Henr 16, Henr 17] is a pure functional data-parallel programming language for nested and array programming that generates OpenCL code for GPU execution. It contains well-known parallel skeletons, such as map, reduce, filter and scan.

Futhark introduced a novel technique to support in-place modification of arrays for the pure functional language, without the need to create a new copy, but only update the modified elements by using a new type feature called *uniqueness type*.

Accelerate [Chak 11, McDo 13] is an embedded programming language integrated for the Haskell programming language. It combines a set of array types to distinguish between Haskell arrays (in the host side) and GPU arrays (in the device side). It also supports a set of operators to execute over the arrays on GPUs such as + or $*$. Accelerate generates a CUDA C kernel at runtime.

Lift [Steu 15, Steu 17] is a data parallel programming language for GPU programming and high-performance execution designed for performance portability across devices. Programmers express high-level parallel operations, such as map and reduce, using the Lift DSL. Then, the Lift program is lowered to an intermediate representation where the runtime explores multiple combination of the input program and apply GPU optimisations by rewriting the input program according to predefined rules.

Dandelion [Ross 13] is a compiler for heterogeneous systems for the Microsoft .NET platform. Dandelion exploits data parallelism using the LINQ language as front-end and its compiler generates CUDA code. Dandelion extends LINQ with new data types for GPUs, therefore, programmers are enforced to change the source code if they want to execute .NET program on heterogeneous systems.

Limitations of explicit parallel programming languages Although explicit parallel programming languages facilitate the compiler with parallelism exploitation, programmers must learn and port the code to the new language. This could be, in many cases, a handicap for many users because of maintaining legacy code. The following sections explore other alternatives to program heterogeneous applications.

3.1.3 Intermediate Representations for Heterogeneous Computing

This section presents intermediate representations (IR) designed for portability and heterogeneous computing. There have been numerous IRs introduced over the last five years to facilitate heterogeneous programming and portability. This section presents the most common ones.

Heterogeneous System Architecture (HSA) [Foun 17a] is a specification developed by the HSA foundation for cross-vendor architectures that specifies the design of a computer platform that integrates CPUs and GPUs. This specification shares the memory among all compute devices (e.g. GPUs) in order to reduce the latencies between them. Besides, the HSA specifies a new intermediate language, called HSAIL

(HSA Intermediate Layer) [Foun 17b]. HSAIL is an explicitly parallel intermediate language across devices which includes support for exceptions, virtual functions and system calls.

Taking the ideas developed in the HSA, AMD developed ROCm [Foun 17c] (Radeon Open Compute Platform), a heterogeneous platform to integrate CPU and GPU management in a single Linux driver for general heterogeneous computing that uses a C/C++ single source compiler. ROCm integrates many projects and many tools under the same compiler and toolchain infrastructure for compiling CUDA, OpenCL and C++ programs automatically.

SPIR [Khro 17] (Standard Portable Intermediate Representation) is a binary intermediate language for compute graphics to express **shaders** (computer programs for processing light, colour, textures of graphics) and compute kernels. SPIR was originally designed by the Khronos Group in order to be used in combination with OpenCL kernels. Instead of compiling the OpenCL source code and distribute it, which can have licensing problems, SPIR defines an intermediate representation based on LLVM in binary format to express the computation on GPUs. SPIR can be used by optimising compilers and libraries to distribute and compile heterogeneous code across multiple heterogeneous architectures.

CUDA PTX [NVID 09] (Parallel Thread Execution) is an intermediate representation developed by NVIDIA for CUDA programs. CUDA is a language and a programming model developed by NVIDIA for NVIDIA graphics cards. CUDA uses a subset of C99 and compiles it to PTX intermediate representation. Then, at runtime, the corresponding CUDA driver compiles the CUDA PTX into machine code for the GPU before executing the CUDA kernel. PTX and CUDA language are only available for NVIDA GPUs.

Jordan *et al.* [Jord 13] presented INSPIRE, a parallel intermediate representation (IR) that uses passes meta-information to indicate to an optimising compiler how to parallelise the code. It uses parallel primitives such as *spawn* and *pfor* among many others in the IR. From the INSPIRE intermediate representation, Jordan *et al.* demonstrated that it is possible to generate OpenMP, MPI, OpenCL and Cilk Plus.

Summary Heterogeneous IRs are not exposed to final users. These IRs are used by optimising compilers and virtual machines to target heterogeneous architectures. This thesis targets OpenCL C as an intermediate language to execute on GPUs because it allows faster prototyping.

3.2 Algorithmic Skeletons

This section presents the most relevant projects that use algorithmic skeletons for heterogeneous programming. Algorithmic skeletons, as Section 2.3 shown, are a programming model for parallel computing. They are presented as high-order functions that facilitates the design and implementation of parallel applications.

Intel Thread Building Blocks (TBB) [Rein 07] is a C++ template library for implementing parallel applications. It exploits task parallelism through a set of parallel operations such as map, reduce and scan as well as primitives to perform thread synchronisation and atomic operations.

Ranger *et al.* proposed Phoenix [Rang 07], a map-reduce parallel execution framework for multiprocessor systems. Similarly, Kovoot *et al.* [Kovo 10] proposed a parallel execution framework based on the Phoenix project for multi-core execution in Java. With these frameworks, applications are written in map-reduce style and the runtime automatically splits the input data and executes the application using multiple threads.

Similarly to Phoenix's work, FastFlow [Dane 15] is a parallel framework that uses a set of algorithmic skeletons for multi-core. Based on FastFlow, Goli *et al.* [Goli 13] proposed a parallel framework for CPU/GPU selection using the **pipeline** and **farm** skeletons in FastFlow and Monte Carlo tree search. It statically analyses which combination is the best for running on CPU and GPU.

PARTANS [Lutz 13] is a parallel framework focused on optimising the data distribution for stencil computation to execute on multiple GPUs simultaneously. For data distribution, PARTANS does not only take the input size and the GPU device information, it also takes the type of PCI express interconnection in the computer system.

LambdaJIT [Lutz 14] is a JIT compiler that optimises skeletons in the C++ Standard Template Library (STL) to execute CUDA code on NVIDIA GPUs. It uses lambda expressions in C++11 as a front end for writing the user defined function and the parallel skeletons from the C++ STL. Then, the LambdaJIT compiler transforms the lambda into a C-struct with a closure function. This transformation is in the AST level. After that, it generates the IR that corresponds to the lambda and optimises it before generating the final code. The final executable contains the generated and optimised IR of the input lambda function. This binary can be used to compile with a CUDA compiler backend based on LLVM.

SkePu [Enmy 10] is a C++ template library that provides a unified interface for programming on CPU and GPUs. It uses parallel skeletons such as map and reduce to generate, at compilation time, the corresponding CUDA and OpenCL kernels. Programmers write the kernels in pure C++ using a set of predefined macros. These macros are expanded to C-structs that contain the CPU-GPU code. The SkePU runtime can also execute the auto-generated GPU program using multiple-devices.

SkelCL [Steu 11, Steu 14] is a C++ library for muti-core and GPU programming that provides a high-level interface based on algorithmic skeletons. SkelCL is built on top of the OpenCL programming model. SkelCL defines a programming interface for data parallelism using parallel patterns such as map, reduce or stencil and its runtime is able to schedule the auto-generated OpenCL program in multiple OpenCL devices at the same time.

Similarly to SkePU and SkelCL, Muesli [Erns 17] is a C++ template library for task and data parallelism that uses parallel skeletons to facilitate the programmability of applications. Muesli is able to parallelise using OpenMP, MPI for shared and distributed systems and CUDA for GPU computing. Muesli-GPU is built on top of CUDA and therefore, it needs a CUDA compiler to compile the whole application. Muesli-GPU also provides an implementation for GPU computing within Java [Erns 15].

Delite [Brow 11, Suje 14] is a compiler framework and a fully embedded DSL for parallel and heterogeneous computing. DSLs allow and facilitate programmers to write code for specific architectures focusing on the algorithm rather than details about how a program should be implemented in order to obtain good performance. Delite DSL uses parallel skeletons such as `map`, `reduce`, `filter`, `groupBy` or `forEach`, among others, to express data parallel operations. Delite DSL is totally embedded into the Scala programming language [Oder 04], but it uses its own intermediate representation in the sea of nodes form [Clic 95]. Programmers write Scala applications using Delite DSL and then they compile them to Java bytecode. Then, the Delite compiler transforms the Java bytecode into the Delite intermediate representation in which Delite applies a set of compiler optimisations. Finally, the Delite compiler compiles the optimised IR to the target parallel programming model (CUDA, OpenCL, Scala or C++) and the Delite runtime executes the generated program.

Summary This approach of programming parallel applications using algorithmic skeletons has been taken for this thesis. As Chapter 4 will show, one of the main contributions is an API based on parallel skeletons for multi-core and GPU programming.

3.3 GPGPU libraries

Another alternative to provide a new language or extensions to the existing ones is via a library that allows programmers to use heterogeneous hardware without modification in the language. There are many libraries for GPU programming. This section describes the most common ones.

CUDA Thrust [Hobe 09, Hwu 11] is a library and an API for array programming using NVIDIA GPU via CUDA in C++. The Thrust library provides an abstraction for the vector types that facilitates the GPU programmability of heterogeneous C++ programs. CUDA Thrust is built on top of CUDA C and CUDA Kernel runtime, therefore, internally, it uses the CUDA C interface to execute on CUDA devices.

In similar way, *cuBLAS* [NVID 12] is an application programming interface developed by NVIDIA that is highly optimised for basic linear algebra domain applications using CUDA. It is implemented on top of CUDA C and CUDA runtime. *clBLAS* is a very similar library to *cuBLAS* but it uses OpenCL instead of CUDA, and it is highly optimised for the AMD hardware.

ArrayFire [Malc 12] is a library for programming heterogeneous applications within C, C++ Fortran Python and Java. It is particularly designed for matrix computation using CUDA and OpenCL and it contains thousands of highly optimised functions on GPUs for signal processing, statistics and image processing.

Java OpenCL (JOCL) [JOCL 17] is a Java binding for OpenCL. JOCL keeps the original OpenCL API into Java using static methods. This approach makes the portability from an OpenCL C program into Java with JOCL very easy by only changing the C specific language parts of the input program into Java specific. The OpenCL kernel remains the same; the kernel is provided as a Java `String` to the driver. Listing 3.2 shows a sketch of the `saxpy` application implemented in Java with JOCL. It only shows the OpenCL kernel (passed as a Java string to the corresponding OpenCL function) and the OpenCL kernel-call in Java. All the OpenCL functions defined in the API are static methods in JOCL. Lines 11-12 invoke the `clEnqueueNDRangeKernel` to execute the kernel on the OpenCL device. All the parameters are Java objects or Java primitives types that are mapped into the OpenCL C via JNI (Java Native Interface) to finally execute the OpenCL code. The final compilation from the Java string to OpenCL C binary is performed by the OpenCL driver.

JogAMD JOCL [Jogamp J 17] is another library that provides a more object-oriented abstraction to program OpenCL within Java. This approach simplifies the programma-

```java
private static final String SAXPY_KERNEL =
   "__kernel void saxpy(__global float *a,\n" +
   "                    __global float *b,\n" +
   "                    __global float *c,\n" +
   "                    const float alpha)\n" +
   "{\n" +
   "    int idx = get_global_id(0);\n" +
   "    c[idx]  = alpha * a[idx] + b[idx];" +
   "}";
// Running the kernel
CL.clEnqueueNDRangeKernel(commandQueue, kernel, 1, null,
                          global_work_size, null, 0, null, kernelEvent);
```

Listing 3.2: Sketch of the `saxpy` application implemented in Java using the JOCL binding

bility of OpenCL C programs, but developers need to know the new API.

JavaCL [JAVACL 15] is another object-oriented approach for OpenCL programming within Java that uses OpenCL across the JNA (Java Native Access) interface to access native shared libraries. JavaCL also provides a low-level interface but this is hidden to the final user.

LWJGL (the Lightweight Java Game Library) [LWJGL 17] includes support for OpenCL, similarly to JOCL. It maps the OpenCL API into Java by using a Java binding.

3.4 Java-Specific GPU JIT Compilers

One of the main contributions of this thesis is focused on programming GPUs from the Java language and automatically generate GPU code from the Java bytecode. This section presents the most relevant academic and industry projects related to Java and JIT compilation for heterogeneous architectures.

3.4.1 Sumatra

Sumatra is a JIT compiler that automatically generates HSAIL [Foun 17b] code at runtime using the Graal compiler for the Java 8 Stream API. Sumatra uses the new Java

```
1 IntStream.range(0, n)   // Create the stream
2          .parallel()    // Set parallel processing
3          .forEach(idx -> {
4              result[idx] = a[idx] * b[idx];  // Computation
5           });
```

Listing 3.3: Vector multiplication example using Java 8 Stream API

8 Stream API to generate code for the accelerator. In particular, it uses the `forEach` method available in Java 8 Streams as a map parallel skeleton to generate code. Listing 3.3 shows a sketch of a Java application that uses the Stream API to multiply two vectors. Line 1 creates the stream using the factory method `IntStream.range()`. Line 2 specifies that the following operations can be computed in parallel. Line 3 invokes the `forEach` for the actual computation. The computation is defined as a Java lambda expression (an anonymous method that can be assigned to a variable).

Sumatra compiles the lambda expression that is passed as argument to the `forEach` method at runtime to HSAIL [Foun 17b]. Sumatra builds a Control Flow Graph (CFG) and a Data Flow Graph (DFG) using the Graal compiler API [Dubo 13b]. Using this CFG as the main data structure for compilation, Sumatra applies a set of compiler optimizations and generates the corresponding HSAIL code for targeted accelerator in the last phases of the Graal compilation pipeline.

The use of `forEach` corresponds to the map parallel skeleton. However, this method returns **`void`**, which does not allow programmers to concatenate with other parallel skeletons.

3.4.2 Aparapi

Aparapi [AMD 16] is a compiler framework for GPU execution via OpenCL. Similarly to Sumatra, Aparapi compiles, at runtime, a subset of the Java program to OpenCL from the Java bytecode.

Aparapi provides an API for programming GPUs within Java. The main Java class in Aparapi is the `Kernel` class. Programmers must provide a program that uses the Apapi `Kernel` class and override its `run` method. The code that defines the `run` method is the code that Aparapi transforms into the main OpenCL C kernel. Aparapi also contains a set of utilities to obtain profiling information and also to force Aparapi to

```
1 public class SaxpyKernel extends Kernel {
2     ...
3     @Override
4     public void run() {
5         int idx = getGlobalId();
6         z[idx] = alpha * x[idx] + y[idx];
7     }
8 }
```

Listing 3.4: Saxpy kernel implemented in Aparapi

compile and execute the Java Aparapi kernel into the GPU. Aparapi kernels correspond to the map parallel skeleton.

Listing 3.4 shows the *saxpy* application implemented in Aparapi. In the run method (lines 4-7), a thread-id is first obtained and stored into a the `idx` variable. This thread-id is a mapping between the Java id and the OpenCL *thread-id* (work-item). Line 6 uses this thread-id (`idx`) to access the arrays and then computes the corresponding values.

All the variables and arrays used in the `run` method have to be declared and allocated before invoking the run method. Otherwise a Java exception is invoked.

Moreover, Aparapi programmers must access arrays using this thread-id in order to map it with an OpenCL thread. Programmer must know, therefore, the OpenCL programming model.

Aparapi JIT compiler takes the Java bytecode that represents the `run` method of the user class and translates it to OpenCL C code. The Aparapi JIT compiler creates a C-struct data structure in OpenCL that corresponds with the object layout of the custom class and a main OpenCL kernel function with the auto-generated run Java method.

At the moment, Aparapi does not support many Java features. It does not support, for example, Java objects, apart from primitives Java arrays, Java exceptions, inheritance or lambda expressions. If an OpenCL kernel can not be generated due to an unsupported feature, Aparapi executes the Java application using Java thread pools. Besides, different kernels can not be composed or concatenated in order to make a pipeline of parallel operations like Java 8 Stream API. Aparapi is limited to build a unique kernel following the `map` semantic. This project, as well as Sumatra, seems a promising approach, although they are not longer maintained.

```
1 public class SaxpyKernel implements Kernel {
2   ...
3   @Override
4   public void gpuMethod(){
5     int index = RootbeerGpu.getThreadId();
6     x[index] += alpha * x[index] + y[index];
7   }
8 }
```

Listing 3.5: Saxpy example implemented with RootBeer framework

3.4.3 RootBeer

RootBeer [Prat 12] is a compiler and framework for GPU computation within Java. Similarly to Aparapi, RootBeer provides an API for heterogeneous programming. The main interface is called `Kernel` and it contains an abstract method (`gpuMethod`) that has to be implemented by the programmer. Similar to Aparapi, the Java code included in this method will be automatically translated to CUDA C code.

In comparison with Sumatra or Aparapi, RootBeer offers a lower-level abstraction of how to program heterogeneous devices. With RootBeer, programmers control in which device/s the code will be executed, create a RootBeer context (similar to OpenCL contexts) and specify how the threads are executed on the GPU. This approach, even though it is in Java, it remains low-level.

RootBeer supports Java objects, Java exceptions, synchronised methods and static as well as dynamic memory allocation on the GPU. To support dynamic memory allocation, RootBeer includes a header for each GPU object that indicate the type of object and implements a heap memory region on the GPU memory. This solution can decrease overall performance on the GPU.

Listing 3.5 shows the *saxpy* implemented in RootBeer. The example only shows the kernel class. The main method to be compiled to CUDA C is the `gpuMethod` in lines 4-7. The `index` variable specifies the thread-id that executes the kernel. This variable is used to access the data.

RootBeer generates CUDA code statically with the Java compiler (`javac`). Programmers implement heterogeneous Java applications using the RootBeer API and then compile the programs using `java` to build a Java package. The code generation is performed via a component called Rootbeer Static Transformer (RST). Programmers

```
1 public class SaxpyKernel implements GPUKernel {
2   ...
3   @Override
4   public void run(){
5     int index = ThreadId.x;
6     z[index] += alpha ∗ x[index] + y[index];
7   }
8 }
```

Listing 3.6: Saxpy example in JaBEE

invoke RST statically with the Java package of the input application. After that, the RST will generate a CUDA kernel and a Java application with the GPU code in a new Java package. This Java package can be directly launched to execute on the GPU. As well as Aparapi, RootBeer does not support function composition or reusability. The CUDA compilation is performed statically with `javac` compiler.

3.4.4 JaBEE

JaBEE [Zare 12] (Java Bytecode Execution Environment) is a compiler framework for compiling Java bytecode to CUDA PTX at runtime, and a execution environment for executing the heterogeneous application on GPUs using CUDA.

JaBEE offers an interface similar to Aparapi and RootBeer. It provides an abstract class (`GPUKernel`) as the main interface with an abstract method, `run`. This method is overwritten in a subclass. JaBEE exposes to programmers the CUDA thread mechanism for accessing data as well as manual synchronisation in the Java code. Then JaBEE compiles, at runtime, all the reachable code accessible from this `run` method. The main differences between JaBEE and Sumatra, Aparapi or RootBeer, is that JaBEE supports objects creation, method dispatch and object communication between host and device. However, the performance is not as good as the other works [Zare 12].

Listing 3.6 shows the *saxpy* application using JaBEE API. The `run` method contains the code to be executed in parallel on the GPU. First, it obtains the thread-id though the static field `ThreadId.x` in line 5. This is equivalent to the `get_global_id(0)` call in OpenCL.

The compilation to the GPU is at runtime via VMKit [Geof 10] using the JIT execution mode. VMKit is a substrate that facilitates the development of high-level lan-

```
1 @Jacc(iterationSpace = ONE_DIMENSION)
2 public void saxpy(@Read float[] x, @Read float[] y, @Read float alpha,
     @Write float[]z) {
3   for (int i = 0; i < x.length; i++) {
4       z[i] = alpha * x[i] + y[i];
5   }
6 }
```

Listing 3.7: Saxpy example in JaCC

guages virtual machines. JaBEE compiles the Java bytecode of the GPU kernel subclass into LLVM Intermediate Representation (LLVM IR) from which the compiler transformations and code generation are applied.

As well as Aparapi and RootBeer, JaBEE can not compose operations in the way Sumatra does. Each kernel has its own separate class with no possibility to concatenate and compose operations. JaBEE execution environment does not perform any data optimization in order to avoid the expensive data transformations between Java and CUDA programming models.

3.4.5 JaCC

Clarkson *et al.* presented the Jacc compiler framework [Clar 17] for parallel and heterogeneous computing. Jacc has its own programming model based on two different types of tasks abstractions: tasks that encapsulate the meta-data needed for GPU execution and tasks-graphs, that define inter-task, control flow and data dependencies. Jacc compiles, at runtime, Java bytecode to CUDA.

The way of programming with Jacc is via Java annotations, similar to OpenACC or OpenMP. Those annotations can be located in the method level, class fields and method parameters. Listing 3.7 shows the *saxpy* example implemented using the Jacc annotation system. Line 1 directs the Jacc compiler to compile the following method to GPU code. In this example, annotations are located in method level trough `@Jacc` annotation. This annotation reports the compiler and the runtime about the iteration space. In this case it specifies that only the outermost loop will be parallelized. Line 2 defines the method to be compiled to the target GPU using CUDA. The parameters to this method are also annotated with either `@Read` or `@Write`. This also reports to the

```
1 public static void saxpy(float alpha, float[] x, float[] y) {
2    for (@Parallel int i = 0; i < y.length; i++) {
3       y[i] += alpha * x[i];
4     }
5 }
```

Listing 3.8: Saxpy example in Tornado

compiler which variables are read-only and which ones are write-only. The following Java code in lines 3-5 contains a sequential Java loop. Jacc compiler transforms this sequential loop into an explicit CUDA parallel loop.

Tornado [Kots 17] is a recent version of the Jacc compiler framework. It reuses the same ideas but it simplifies the API, reducing also the number of annotations needed to express a parallel code. Listing 3.8 illustrates the *saxpy* application using the Tornado framework. Line 2 uses the Java annotation `@Parallel` that identifies that the index *i* corresponds to a parallel loop. Then the compiler builds the CFG for the whole method and analyses its data dependencies.

One of the key limitations with the annotation approach is that parallel operations can not be composed and concatenated the way Sumatra does. Jacc also provides the creation of tasks and task-graphs where it should be possible to compose and create a graph. However, although this is programmed in pure Java, it still remains low-level because it requires the programmer's expertise to properly create the task dependency graphs.

3.4.6 IBM J9 GPU

IBM J9 GPU [Ishi 15] is a GPU compiler framework based on Java 8 Stream API that compiles, at runtime, the Java bytecode to CUDA PTX. This project, is, in fact, very similar to Sumatra.

Listing 3.9 shows a Java application to solve the *saxpy* in IBM J9 in order to use the GPU. This example is, in fact, the same as Sumatra, because it does not use any additional interface or library. Lines 4-6 create the stream and invoke the `forEach`.

The JIT compiler translates the Java bytecode into another intermediate representation (IR), then applies optimizations to the Java program and then it searches for the parallel method `forEach`. Then the `forEach` is compiled into NVVM IR (Nvidia

```
float[] a = ... ;
float[] b = ... ;
final float alpha = 2.1f;
IntStream.range(0, n)
        .parallel()
        .forEach( idx -> alpha * a[idx] + b[idx]);
```

Listing 3.9: Saxpy example in IBM J9 GPU

intermediate representation based on LLVM IR).

IBM J9 uses the `forEach` as the entry door for parallelisation on GPUs. This operation in the Java 8 Stream API is a terminal operation that can be concatenated and reused again with other Java streams. Moreover, IBM J9 does not optimise the data transformation between Java and OpenCL and it can not dynamically create new arrays for known size.

3.4.7 Summary

This section has introduced the main Java related works on GPU compilation. Table 3.1 summarises important features for all discussed frameworks. The first column shows each of the Java projects introduced in this section. The rest of the columns indicate Java features and OpenCL or CUDA optimisations for executing on GPUs. The second column indicates if the compilation is performed at runtime (JIT). The third column indicates if data is optimised between Java and CUDA or OpenCL to avoid marshalling. The fourth column shows whether specific memory optimisations are performed, such as OpenCL pinned memory. The firth column indicates if dynamic array allocation is supported. The sixth column indicates if any Java objects are supported. The seventh column indicates if there is a mechanism to fall back the execution in case there is an unexpected runtime error on the GPU. The last column indicates if the related work supports lambda expressions.

None of these works contains all the summarised features in the Table 3.1. The most complete ones are JaBEE and JaCC. However they still do not perform optimisations of data transformations between Java and OpenCL, or support lambda expressions. This thesis proposed a system that contains all of these features, as Chapters 4 and 5 will show.

Work	JIT	Data Layout Opt.	Transfer Opt.	Array Alloc.	Java Objects	Fall back to Java	Lambda
Sumatra	Yes	No	No	Yes	No	No	Yes
JOCL	No	Manual	Manual	No	No	No	No
Aparapi	Yes	No	No	No	No	Yes	Yes
[Ishi 15]	Yes	No	Yes	No	Yes	Yes	Yes
RootBeer	No	No	No	Yes	Yes	No	No
JaBEE	Yes	No	Yes	Yes	Yes	No	No
JaCC	Yes	No	Yes	Yes	Yes	Yes	No

Table 3.1: Features comparison for GPU programming in Java

3.5 Language Implementations for Truffle

As seen in Section 2.4.4, Truffle is a DSL framework to implement programming languages on top of the JVM. Truffle has a wide variety of language implementations ranging from functional and technical computing, pure object oriented programming languages to low-level IR.

For technical computing, Truffle currently supports R [Kali 14, Stad 16] (called **FastR**) and J programming languages [Imam 14]. J is a language based on APL [Hui 90] in which arrays are the main data types for computation.

There are also Truffle implementations for low level programming languages like C, called TruffleC [Grim 14] and a lower-level implementation for the LLVM IR [Rigg 16]. LLVM (Low-Level Virtual Machine) [Latt 02] is a tool-chain and a compiler framework to implement compiler front-ends and back-ends. LLVM describes an intermediate representation called LLVM IR and it is based on triplets code in SSA form. LLVM has multiple front-ends, such as Fortran, C or C++. Sulong (Truffle LLVM) will allow the execution of these low-level programming languages on top of the JVM using the Graal compiler infrastructure.

There is also an implementation for SOM SmallTalk [Marr 15] (a smaller SmallTalk version designed for teaching and researching on VMs),

Truffle also supports dynamically typed programming languages such as Python, called Zyppy [Wimm 13], Ruby, called Truffle-Ruby [Seat 15, Dalo 16] and Javascript, which is called GraalJS [Leop 15, Wurt 17].

Summary Currently, none of these high-level languages can execute code on GPUs. One of the main contributions of this thesis is to provide an OpenCL JIT compiler for a dynamic typed and interpreted programming language. As a use-case, all of the compiler techniques proposed in this thesis have been implemented using the R programming language for Truffle (FastR).

3.6 GPU Programming for Dynamic Programming Languages

This section reviews the relevant literature on exploiting parallelism from dynamic programming languages to use GPUs. It first discusses the works related to the R programming language for GPU execution since one of the main contributions of this thesis uses R as a use-case. The last part of this section presents related works for other dynamic and interpreted programming languages.

3.6.1 Library-based R GPU Support

The majority of approaches for parallelizing programs written in dynamically interpreted languages, such as R, relies on the use of libraries. There are numerous R libraries that support CUDA and OpenCL execution. These libraries typically implement well-known functions for specific application domains, such as the *GPUR*[1] library which supports matrix operations.

Internally, these libraries are implemented in CUDA or OpenCL, or leverage an existing implementation, such as ViennaCL for GPUs [Kyse 12] through an API. Other approaches rely on the use of wrappers for low-level interface such as OpenCL and they require the programmer to write OpenCL code inside R programs. The compiler technique presented in this thesis is fully automatic and it does not rely on any library implementation. Moreover, it is more generally applicable than all these existing library-based approaches.

GMatrix and *GPU Tools* are libraries for accelerating R matrix operations with CUDA. They implement common vector and matrix operations such as matrix multiplication, addition, subtraction, sorting and trigonometric functions. With GMatrix, the programmer explicitly creates and manipulates GPU objects in R.

[1]`https://cran.r-project.org/web/packages/gpuR/index.html`

Rth[2] is an interesting project which provides an R interface for CUDA Thrust. The package executes parallel code for three different back-ends for the same interface: CUDA, OpenMP and Intel TBB.

OpenCL for R[3] is a wrapper that exposes the OpenCL API to R. However, it changed the standard OpenCL API compared to the OpenCL standard. Thus programmers need to learn a new non-standard API. The user also provides the OpenCL kernel as a string in R that will be compiled by the OpenCL driver. This is, in fact, a very low-level approach where the parallelism is fully exposed to the R programmers. Moreover, because of the OpenCL kernel is expressed in R as a string, it increases the complexity and manageability of the code.

There are other projects specialized for statistic computation on GPUs. *RPUD* and *RPUDPLUS*[4] are open source R packages for performing statistical computation using CUDA. RPUD implements vector operations and Bayesian classification algorithms. In this case, R programmers import the library and uses the GPU with predefined functions. Those functions call directly to the GPU.

3.6.2 Parallelism Exploitation in R

Riposte [Talb 12] uses trace-driven runtime compilation to dynamically discover vector operations from arbitrary R code and it produces specialized parallel code for CPUs. Riposte exploits vector operations on multi-core CPUs.

Wang et. al [Wang 15] have targeted the `apply` function to extract parallelism by automatically vectorizing [Wang 15] its implementation for multi-core CPUs. Wang shows an algorithm to vectorize R code and transforms the looping-over data functions for the apply methods in the R interpreter.

There has been work on using specialisation techniques with R in order to speed up the execution of the R interpreter. FastR [Kali 14, Stad 16], on which this thesis is based, is a Truffle implementation that uses Graal as a JIT compiler to produce efficient code for the CPU.

Wang et. al [Wang 14a] introduced Orbit VM, an R VM that uses profile-directed specialisation techniques for accelerating the execution of R code. Orbit VM accelerates the current GNU interpreter by specialising types and objects at runtime. Wang's work reduced the interpreted overhead.

[2]`http://heather.cs.ucdavis.edu/~matloff/rth.html`

[3]`https://cran.r-project.org/web/packages/OpenCL/`

[4]`https://cran.r-project.org/src/contrib/Archive/rpud/`

Renjin [Kall 15] is an R implementation of top of the JVM. Renjin automatically parallelises vector operations using multiple threads in Java. It uses a delay computation mechanism similar to lazy evaluation that produces a computation graph. When the results are needed, the computation is then optimised using type information available, for instance to produce a more efficient execution of the computation.

Summary These works have targeted multi-core CPUs to exploit primitives in the R language. However, none of them have targeted GPUs. The work presented in Chapter 6 of this thesis is the first that performs automatic GPU JIT compilation and runtime management for the R programming language without the need of any additional library or interface.

3.6.3 GPU Acceleration for other JIT Programming Languages

Very few GPU code generators exist for other interpreted languages. *Numba* [Lam 15], already mentioned in Section 3.2 is a CUDA JIT compiler for python which is based on annotations to identify parallel sections of code and data types.

Springer *et al.* [Spri 17] presents *Ikra*, an array-based dynamically typed language to execute on GPUs as an extension of Ruby. Ikra library can handle polymorphic types at the cost of performance. Besides, programmers are required to change the source code to include the Ikra data types.

RiverTrail [Herh 13] is defined as a parallel programming model and an API for the JavaScript language to execute on GPUs using OpenCL. It introduces a new data type, named `ParallelArray`, and a set of new parallel functions such as map, reduce, scan, filter and partition over the arrays. They are exposed to programmers to be used in combination with the new array parallel type. River Trail has a compiler that performs type inference from the untyped JavaScript to the typed OpenCL and generates the OpenCL C code to execute on GPUs. With River Trail, the parallel code is identified by using the `ParallelArray` type with its parallel operations.

Similarly to Rivertrail, ParallelJS [Wang 14b] is a JavaScript parallel execution framework to execute on heterogeneous systems with GPUs. ParallelJS includes custom data types that programmers need to use and a set of parallel operations on those arrays, such as map, reduce, filter, etc. Profiled type is performed dynamically, and monomorphic types are assumed with no recovering mechanism in the case of the type inference is not performed correctly.

Harlan-J [Pita 13] is an OpenCL JIT compiler for Javascript. It is based on Harlan-J language, a Javascript extension for data parallelism. With Harlan-J programmers create arrays using the `ArrayBuffer` type. These arrays are passed to the kernel functions.

Summary None of these approaches provides a fully automatic heterogeneous JIT compiler for high-level languages. Programmers need to change the code and adapt it for GPU execution. As Chapter 6 will present, this thesis shows how to compile, at runtime, a program written in R to OpenCL with no changes in the source code.

3.7 Summary

This chapter has presented the most relevant works for this thesis. It has focused on data parallelism and GPU compiler frameworks for high-level programming languages. It has emphasized the works related to algorithmic skeletons and JIT compilation for Java, interpreted programming languages to GPU code.

Prior works have facilitated the programmability of GPUs. However, most of them have been focused either on low-level programming languages, such as C, C++, new DSLs, or new programming languages. Little work have been proposed on exploiting existing high-level and dynamically typed programming languages for GPU computing. To exploit GPUs from interpreted programming languages such as Java or R, not only the GPU code is important for performance, but also optimising data transformations between high-level and low-level languages. This is critical task in order to achieve a good overall performance. None of the works presented in this chapter have looked, in detail, to the code generation and runtime data management for efficient GPU computation from Java or R languages. The next three chapters present a solution to use GPUs from interpreted programming languages and they discuss, in detail, how this is achieved.

Part II

CONTRIBUTIONS

Chapter 4

A Composable and Reusable Array API for Heterogeneous Computing

Heterogeneous computing has now become mainstream with virtually every desktop machine featuring accelerators such as GPUs. GPUs are attached to almost every computer and mobile system, for example computers (PCs), servers, mobile devices and ad-hoc hardware for specific purpose. Programming such systems requires deep understanding of the underlying technology and new parallel programming models (such OpenCL explained in Section 2.2). However, many consumers of GPUs, such as economists, biologist, etc, are not programmers or specialists on parallel programming and computer architecture. These users tend to program in high-level programming languages like Java. There is a wide variety of abstractions for low-level and unmanaged programming languages like C/C++. However, little attention has been given to provide high-level abstractions for parallel and heterogeneous programming for managed programming languages like Java.

This chapter presents and describes a new Java Parallel Array Interface (JPAI) for programming heterogeneous architectures within Java. JPAI allows developers to execute Java programs on GPUs and CPUs transparently with the same programming interface and without any change of the source code. Composability and reusability are the two key features of JPAI compared to the state-of-the-art in related Java APIs for programming accelerators. This chapter is focused on the design and implementation of JPAI and its evaluation on a multi-core CPU.

This chapter is structured as follows. Section 4.1 motivates this chapter and introduces the main terms. Section 4.2 presents and discusses the implementation of the proposed JPAI. Section 4.3 shows how compute operations are composed in JPAI.

Section 4.4 presents how JPAI operations are executed and Section 4.5 shows how JPAI operations can be reused multiple times. Section 4.6 presents a set of applications written in Java with JPAI. Section 4.7 evaluates the performance of JPAI using a multi-core CPU. JPAI is also able to execute parallel operations on GPUs. The next chapter will present, discuss and evaluate, in detail, the GPU implementation of JPAI. Finally, Section 4.8 summarises the main contributions presented in this chapter.

4.1 Motivation

As discussed in Chapter 2, there are several languages and frameworks for programming GPUs. The most common and extended ones are OpenCL and CUDA already introduced in Section 2.1. Writing efficient parallel code using these models requires a deep understanding and a good knowledge of the underlying hardware. Because OpenCL and CUDA are very low-level programming models, they force programmers to control the location of every single buffer, kernel creation and compilation, synchronization, thread scheduling, etc. This detailed level of control allows developers to take advantage of many resources available in the computer system. However, this fine-control increases the programmability and software complexity.

Languages and interfaces for programming these kinds of systems tend to be low-level and require experts' knowledge of the GPU architecture. Programming in low-level programming languages and models is a hard task for non-expert programmers. However not all the scientists are, or have to be, experts in software and parallel computer programming. Thus, it is important to raise program abstractions for non-expert programmers, so they can also exploit the potential of parallel accelerators.

Use of Parallel Skeletons for GPU Programming A better approach consists of using structured parallel programming where common patterns are used to easily express algorithmic ideas as parallel operations on arrays (see Chapter 3.2). This style of programming is heavily influenced by functional programming and has been successfully applied to practical use, e.g., in Google's MapReduce framework [Dean 04]. Structured parallel programming helps to simplify the programming of parallel hardware and, at the same time, it enables the compiler to produce efficient parallel code by exploiting the semantic information from the parallel patterns. As Chapter 3 showed, this idea is not new and it has already been applied to GPUs by high-level libraries and languages and compilers. Most projects have targeted either functional programming

```
1 int result = IntStream.range(0, n)                // Create the stream
2                        .parallel()                // Set parallel processing
3                        .map(idx -> a[idx] * b[idx])// Computation map
4                        .reduce( (i, j) -> i + j)  // Computation reduce
5                        .getAsInt();               // Get int value
```

Listing 4.1: Example of a Java program using the Stream API for `dotProduct`

languages or languages that are common in the high performance computing community, such as C and C++. Only little work has been done to integrate these ideas for programming GPUs into existing mainstream object oriented languages like Java. The challenge is to design a system which combines the familiarity of object oriented languages with the power and flexibility of parallel patterns to enable Java programmers to use GPUs.

One of the main works in the Java world is the Java Stream API introduced in Java 8 for multi-core CPUs, and Sumatra, that uses the same Stream API for GPU execution. Stream API allows programmers to create streams from Java collections and to manipulate them on multi-core CPUs. The API uses well-known parallel skeletons such as `map` and `reduce`. Listing 4.1 shows an example of a Java program using the Stream API to solve the *dot product* problem. Line 1 creates a Java stream; line 2 sets the stream to be processed in parallel. Line 3 calls the map function to multiply the two vectors and line 4 applies the reduction to sum up all the elements in the array. Note that the two arrays (`a` and `b`) are introduced via lexical scope to the lambda expression. Finally, line 5 gets the result as a single integer value.

Limitations of Existing Works Java Stream API is a good interface that simplifies the manipulation and data processing of Java collections and arrays. However, it has some important limitations. A key limitation of Stream API is that the input function can not be reusable. The computation is tightly associated to the input stream (input data). If programmers invoke again the same computation but with different data, the stream is recomputed. A consequence of this limitation is that the stream has to be rebuilt again, even if the same computation is performed. If the computation is very compute-intensive, the Java Virtual Machine (JVM) can compile the input computation to efficient machine code. However, if a new stream is built for the same computation, the JVM can not figure out that the computation is the same (because it is a new stream), and therefore, take advantage of the already profiled and compiled machine code.

Another important limitation is that programmers have to invoke the `parallel()` method manually. This method informs the framework that the upcoming set of operations can be computed in parallel. Therefore it is responsibility of the programmer to make the static decision of knowing if the input stream can be processed in parallel or not.

Solution: Composable and Reusable API for Heterogeneous Programming To overcome these limitations, this thesis proposes a Java Array Programming Interface (JPAI). JPAI is a new Java API based on Java 8 Stream for parallel and heterogeneous programming. JPAI uses algorithmic skeletons and the new feature of Java lambda expressions to facilitate the programmability and readability. Composition and reusability of parallel skeletons are the two key features of JPAI.

This chapter makes the following contributions:

- Design of JPAI, a new API for supporting array programming in Java.
- Set of scientific applications implemented with JPAI.
- Evaluation of JPAI for multi-core computer systems.

4.2 JPAI: Array Programming in Java

Inspired by the Java Stream API, this chapter presents a **Java Parallel Array Interface** (JPAI) for parallel and heterogeneous computing within Java. JPAI implements well-known parallel skeletons such as `map` and `reduce` operations for array parallel programming. These operations are defined independently of the input array instances.

JPAI is a simple API that is easy to use and to learn, and it allows composability and reusability of parallel skeletons. JPAI has two different execution modes; one for multi-core via Java threads and another one for GPU execution. It uses the same programming interface for both execution modes. The JPAI runtime is the software component that decides where to execute the code. This chapter is focused on the multi-core execution mode. The accelerator mode for the GPU execution is discussed in Chapter 5.

JPAI uses the new functional interface (`@FunctionalInterface`) introduced in Java 8 to facilitate the programmability. A Java `@FunctionalInterface` is an interface with

```
1 int[] x = new int[n];
2 int[] y = new int[n];
3 initialization(x, y);
4 int acc = 0;
5 for (int i = 0; i < x.length; i++) {
6         acc += (x[i] * y[i]);
7 }
```

Listing 4.2: Dot product code in Java

exactly one abstract Java method. JPAI allows programmers to override the abstract method using a more concise syntax through Java lambda expressions. Lambda expressions are particularly useful to reduce the boilerplate code and make the program easier to understand. In this way, a Java program written in JPAI is easier to read and maintain.

JPAI allows programmers to focus on the algorithm rather than details of how to obtain good performance. JPAI relies on a runtime system that parallelizes and manages the application automatically. This section presents the design and implementation of JPAI in order to enable parallel and heterogeneous programming within Java with minimal effort. The section starts by showing an example of JPAI in Java and describing the different parts of the API. Then, all the supported operations and their internal design are presented.

4.2.1 Example: Dot product

This section introduces JPAI through a very simple example that shows the main parts of the API and how the computation is expressed.

The *dot product* computation of two vectors is defined as follows:

$$dotProduct = \sum_{0}^{(n-1)} x_i * y_i$$

The dot product is computed by multiplying the two vectors pairwise and then by summing up all intermediate results. Listing 4.2 shows the corresponding Java sequential code for the *dot product*. Lines 1-2 create the input data. Lines 5-7 iterate over the input arrays to multiply and accumulate the result in `acc`. The final result is in the accumulator (`acc`).

```
1  ArrayFunction<Tuple2<Float, Float>, Float> dotProduct;
2  dotProduct = new Map<>(vectors -> vectors._1 * vectors._2)
3                .reduce((x, y) -> x + y, 0.0f);
4  PArray<Float> result = dotProduct.apply(input);
```

Listing 4.3: Dot product Java code with `ArrayFunction` API

Listing 4.3 shows the *dot product* computation using JPAI. The JPAI main interface is the `ArrayFunction` Java interface, which directly inherits from the `Function` interface in Java 8. `ArrayFunction` declares the methods `apply`, `map`, `reduce` among others. It will be discussed in detail in the next sections.

JPAI is designed for array programming. Thus, when declaring a user function with JPAI, the specified input and output types correspond to arrays of the given type. Line 1 of Listing 4.3 declares a function that computes from an input array of `Tuple2` to an output array of type `Float`. A `Tuple2` object contains two fields (`_1` and `_2`) that are used to store an element of input arrays (one field per array). Since the apply method of the Java `Function` interface receives only one input, multiple inputs are packed into Tuples. This is the case of the `dotProduct` example, which perform a computation from two input arrays. Line 2 of Listing 4.3 invokes the map function that multiplies the two vectors that are contained in the fields `_1` and `_2` of the tuple. Line 3 invokes the `reduce` function in a similar way to the Stream API. This operation sums all the elements produced in the previous map operation. Lines 2-3 do not execute the operations but they only define the computation. Only when the `apply` method is invoked, in line 4, the expression is executed. The result is stored in a **PArray** object (*Portable Array*) which is a custom and optimized array for optimising data transformations between Java and the GPU. This object is the basic input and output type in JPAI and it is specially designed for parallel processing on accelerators. The main reason of having a custom object for input and output representation is for optimizing the data transformation between Java and accelerators. Details of the `PArray` will be covered in Chapter 5.

This example illustrates how to program functions with JPAI. One of the key features of JPAI is reusability. Listing 4.4 shows an example of how JPAI reuses the same function for different input data sets. The `dotProduct` function is the same array function that was declared in Listing 4.3 and it is now executed with different input data.

```java
// Other input data
PArray<Tuple2<Float, Float>> otherInput = ...;
// Compute same function with the new input data
PArray<Float> otherOutput = dotProduct.apply(otherInput);
```

Listing 4.4: Dot product Java code with `ArrayFunction` API defined in 4.3 using a different input data set.

4.2.2 JPAI Design

This subsection presents the design and the implementation of JPAI. As mentioned in the motivation section 4.1, JPAI is inspired by the Stream API and uses lambda expressions which are part of Java 8. However, in contrast to the Stream API, JPAI implements all array operations as functions which inherit from the same functional interface. This enables the creation of reusable composed functions, which is fundamentally different from the Stream API, where parallel operations are invoked according to one particular instance of the input data.

Figure 4.1 shows a simplified view of the class hierarchy in JPAI in the Unified Modeling Language (UML) form. Colours represent different categories of parallel skeletons. For example, all the *map* parallel skeletons are represented in green. JPAI provides different versions for each pattern depending on the underlying technology (e.g. OpenCL or Java threads). This section explains each of these categories and their variants.

UML representation UML is a standardised form of expressing class relationship in object oriented programming languages. Each box in an UML diagram represents a class or an interface. Each box is divided in three sections: the top section specifies the class or interface name; the second section specifies the fields of the class and the bottom section specifies the methods. Symbols before the signature of each method and fields in the UML diagram indicate the visibility. Symbol "+" indicates that a method or field is public whereas "#" indicates that a method or a field is protected.

Array Function Design The core interface is the Java `Function` functional interface in Java 8. This interface is shown at the top of the Figure 4.1. It contains exactly one abstract method, `apply`, to be implemented in the inherited classes. This is the core method for of all operations. Each parallel skeleton (e.g. map, reduce) implements its

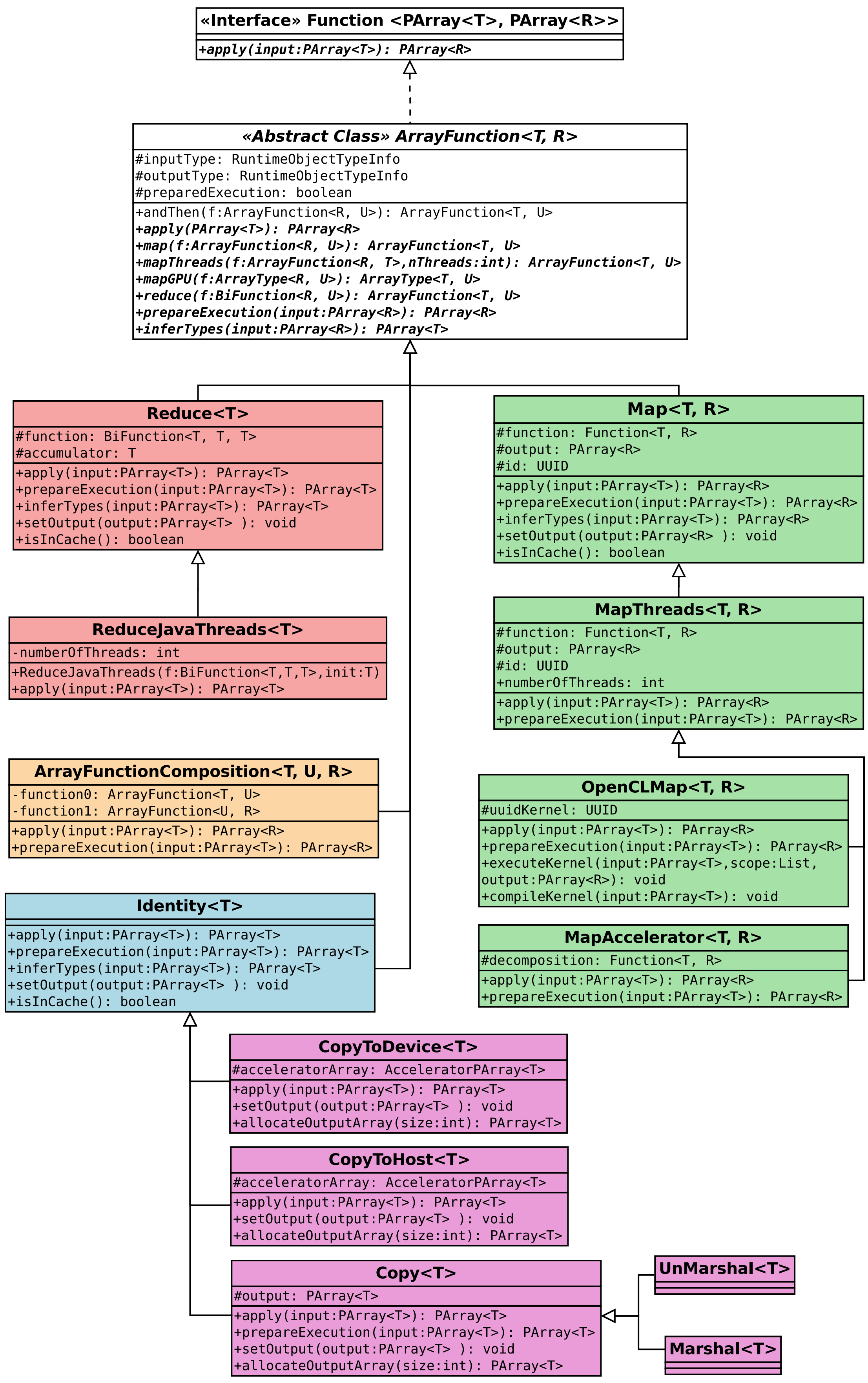

«Interface» Function <PArray<T>, PArray<R>>
+apply(input:PArray<T>): PArray<R>
«Abstract Class» ArrayFunction<T, R>
#inputType: RuntimeObjectTypeInfo
#outputType: RuntimeObjectTypeInfo
#preparedExecution: boolean
+andThen(f:ArrayFunction<R, U>): ArrayFunction<T, U>
+apply(PArray<T>): PArray<R>
+map(f:ArrayFunction<R, U>): ArrayFunction<T, U>
+mapThreads(f:ArrayFunction<R, T>,nThreads:int): ArrayFunction<T, U>
+mapGPU(f:ArrayType<R, U>): ArrayType<T, U>
+reduce(f:BiFunction<R, U>): ArrayFunction<T, U>
+prepareExecution(input:PArray<R>): PArray<R>
+inferTypes(input:PArray<R>): PArray<T>
Reduce<T>
#function: BiFunction<T, T, T>
#accumulator: T
+apply(input:PArray<T>): PArray<T>
+prepareExecution(input:PArray<T>): PArray<T>
+inferTypes(input:PArray<T>): PArray<T>
+setOutput(output:PArray<T>): void
+isInCache(): boolean
Map<T, R>
#function: Function<T, R>
#output: PArray<R>
#id: UUID
+apply(input:PArray<T>): PArray<R>
+prepareExecution(input:PArray<T>): PArray<R>
+inferTypes(input:PArray<T>): PArray<R>
+setOutput(output:PArray<R>): void
+isInCache(): boolean
ReduceJavaThreads<T>
-numberOfThreads: int
+ReduceJavaThreads(f:BiFunction<T,T,T>,init:T)
+apply(input:PArray<T>): PArray<T>
MapThreads<T, R>
#function: Function<T, R>
#output: PArray<R>
#id: UUID
+numberOfThreads: int
+apply(input:PArray<T>): PArray<R>
+prepareExecution(input:PArray<T>): PArray<R>
ArrayFunctionComposition<T, U, R>
-function0: ArrayFunction<T, U>
-function1: ArrayFunction<U, R>
+apply(input:PArray<T>): PArray<R>
+prepareExecution(input:PArray<T>): PArray<R>
OpenCLMap<T, R>
#uuidKernel: UUID
+apply(input:PArray<T>): PArray<R>
+prepareExecution(input:PArray<T>): PArray<R>
+executeKernel(input:PArray<T>,scope:List,
output:PArray<R>): void
+compileKernel(input:PArray<T>): void
Identity<T>
+apply(input:PArray<T>): PArray<T>
+prepareExecution(input:PArray<T>): PArray<T>
+inferTypes(input:PArray<T>): PArray<T>
+setOutput(output:PArray<T>): void
+isInCache(): boolean
MapAccelerator<T, R>
#decomposition: Function<T, R>
+apply(input:PArray<T>): PArray<R>
+prepareExecution(input:PArray<T>): PArray<R>
CopyToDevice<T>
#acceleratorArray: AcceleratorPArray<T>
+apply(input:PArray<T>): PArray<T>
+setOutput(output:PArray<T>): void
+allocateOutputArray(size:int): PArray<T>
CopyToHost<T>
#acceleratorArray: AcceleratorPArray<T>
+apply(input:PArray<T>): PArray<T>
+setOutput(output:PArray<T>): void
+allocateOutputArray(size:int): PArray<T>
Copy<T>
#output: PArray<T>
+apply(input:PArray<T>): PArray<T>
+prepareExecution(input:PArray<T>): PArray<T>
+setOutput(output:PArray<T>): void
+allocateOutputArray(size:int): PArray<T>
UnMarshal<T>
Marshal<T>

Figure 4.1: Simplified view of the class hierarchy of the `ArrayFunction` API. Each colour represents a category of parallel skeleton.

own version of the `apply` method. JPAI implements the Java `Function` interface using the type `PArray`. `PArray`, as briefly explained in Section 4.2.1, is a new data structure used by JPAI for efficient array computation.

The `ArrayFunction` class extends the Java `Function` interface and is intended to be used as the top-level interface by the programmer. The main class is `ArrayFunction`, and as its name suggests, this class represents functions that operate on arrays. The elements of the input arrays are of type `T` while the output elements are of type `R`. This means that `ArrayFunction` processes arrays from `PArray<T>` to `PArray<R>`.

Each parallel skeleton inherits from the `ArrayFunction` class and each operation returns a new `ArrayFunction`. When an operation such as `map` is invoked, a new instance of the `ArrayFunction` class is created, allowing composition of other parallel skeletons.

Array Operations The `ArrayFunction` class comprises several methods corresponding to the different available functions operating on arrays. This class contains methods such as `map`, `reduce` or `pipeline` and a set of methods for parallel computing such as `mapThreads` and `mapGPU` as can be seen in Figure 4.1. These methods create `ArrayFunction` subclasses that implement the different operations. The `Map` class implements the sequential version for the parallel map skeleton. There are also two more versions for parallel execution with multi-core via Java threads and GPU via OpenCL. The `mapThread` operation creates a class `MapJavaThreads` and, when the apply method is invoked, it computes the map using Java threads. In similar way, the `MapAccelerator` implements the map for OpenCL. The apply method in `MapAccelerator` compiles, at runtime, the user function to OpenCL C and executes it on an OpenCL device.

JPAI also provides array operations for copying data from and to the OpenCL device. The main motivation for this is that, in order to execute code on GPUs, arrays need to be firstly allocated on the GPU, and secondly transferred to the GPU, as explained in Section 2.2.

Summary This section has presented a high-level overview about the operations that JPAI implements. The following sections explain each group of operations in detail. It starts by describing the map and copy operations and then the reductions.

4.2.3 Map Operations

This section presents all the different implementations for the map skeleton available in JPAI. This corresponds to the green blocks in the UML diagram in Figure 4.1 (`Map`,

```
for (int i = 0; i < n; i++) {
    output[i] = f(input[i]);
}
```

Listing 4.5: Sequential implementation of the map parallel skeleton

MapThreads, OpenCLMap and MapAccelerator). Apart from the original Map<T,R> version, none of these versions of the map operation are exposed to the final user. This means that programmers only use the basic map version and the JPAI runtime automatically rewrites the expression, before executing in the desired hardware, into one of the concrete map instances. When an ArrayFunction is built, the constructor inspects if there is an OpenCL device available. In that case, JPAI rewrites the expression to use the MapAccelerator function, otherwise it uses the MapThreads version.

The *map* parallel skeleton, as explained in Chapter 2.3, is an algorithmic skeleton that applies a single function to a set of input elements as shown in Listing 4.5. The input and output for the map computation are arrays. The map parallel skeleton has no data dependency between iterations.

JPAI implements different versions of the map skeleton. As Figure 4.1 shows, JPAI implements four different versions.

Sequential map This class implements directly the sequential implementation. This is used for debugging and for applying deoptimisations [Holz 92]. Deoptimisations means that the runtime system discards the optimised version for the non-optimal if a miss-speculation or a problem is encountered and falls back the the slow and non-speculative implementation. Deoptimisations can happen due to unsupported OpenCL features and OpenCL exceptions. In this way, JPAI always guarantees that the input program is executed.

Listing 4.6 shows the apply implementation of the sequential map parallel skeleton. All operations follow the same structure. Firstly, it computes the first element from input data set to infer the types in line 4 (when the method prepareExecution is invoked). This is because JPAI uses Java generics. Java types are erased at runtime and it needs to be executed in order to obtain the type, which is required for allocation of the output array. Secondly, JPAI allocates the output in line 7, and finally it computes the map in a sequential **for** loop in lines 9-11.

```java
@Override
public PArray<R> apply(PArray<T> input) {
    if (!preparedExecutionFinish) {
       prepareExecution(input);
    }
    if (output == null) {
        output = allocateOutputArray(input.size(), input.getStorageMode());
    }
    for (int i = 0; i < input.size(); ++i) {
        output.put(i, function.apply(input.get(i)));
    }
    return output;
}
```

Listing 4.6: Sequential Map algorithmic skeleton implementation in JPAI

Multi-Thread Map JPAI implements a multi-thread version of the map parallel skeleton via the `MapThread` class. The way JPAI implements it is very straightforward: the apply method receives the function, the input data and the number of partitions. The JPAI multi-thread implementation splits the iteration space into the number of partitions (which is normally the number of machine cores) and processes one of the chunks by each thread.

OpenCL Map `OpenCLMap` compiles, at runtime, the User Defined Function (UDF) to OpenCL and execute the generated OpenCL program on the GPU. Its `apply` method contains the logic for performing:

1. OpenCL JIT Compilation of the input function.
2. OpenCL execution of OpenCL code on the GPU.
3. Runtime data optimizations between Java and OpenCL that improves the overall computation with execution on GPUs.

The implementation of this version of the map parallel skeleton is slightly more complex than the other versions. This is because JIT compilation and GPU execution have to be performed. In overview, this map implementation creates, at runtime, the

Java bytecode that corresponds to the input function to be compiled, performs a set of compiler optimizations and generate the OpenCL C code.

For compiler optimizations, JPAI uses a new Java JVM JIT compiler (Graal) and extends it for OpenCL compilation. The `apply` method invokes the OpenCL JIT Compiler (called *Marawacc*) and uses the Marawacc runtime to orchestrate and optimise the data transformation when running on GPUs.

The `apply` method of this parallel skeleton only contains the logic for the execution on the GPU, not the data transfers from the the host to the device and vice-versa. This version of the map parallel skeleton has to be combined together with the `HostToDevice` and `DeviceToHost` skeletons. To simplify the programmability, JPAI also provides the skeleton `AcceleratorMap`, which internally invokes all of these operations (`CopyToHost`, `OpenCLMap` and `CopyToDevice`).

Accelerator Map Sequential map, multi-thread and OpenCL constitute the core versions of the map pattern for different parallel hardware. JPAI implements a fourth one, called `AcceleratorMap` as Figure 4.1 shows. This version uses function composition in JPAI (that will be explained in detail in the following section), to facilitate the programmability with OpenCL.

In order to execute in OpenCL, first, the data is transferred from the CPU to the GPU, then the kernel is executed and finally the data is transferred back from the GPU to the CPU. JPAI decomposes the original `AcceleratorMap` expression into a new expression composed by copying in, execution and copying out. The `OpenCLMap` class corresponds to the execution step. `AcceleratorMap` composes these three expressions in one single operation using the `andThen` method from the Java `Function` interface.

Listing 4.7 shows a sketch of the `AcceleratorMap` operation. Lines 1-6 correspond to the constructor, where the function uses composition to build a sequence of operations for copying to the device, execution and copying the data to the host. The `apply` method in lines 7-13 simply calls the `composition apply` directly, reducing boiler plate code.

4.2.4 Reduce Operations

Reduction and map skeletons are the two most important algorithmic skeletons. Many applications can be expressed with these two parallel skeletons. A reduction operation is defined as a combinator of multiple elements from an input collection into a summary value.

```java
public MapAccelerator(Function<T, R> function) {
    CopyToDevice<T> copyToDevice = new CopyToDevice<>();
    OpenCLMap<T, R> openclMap = new OpenCLMap<>(function);
    CopyToHost<T> copyToHost = new CopyToHost<>();
    composition = copyToDevice.andThen(openclMap).andThen(copyToHost);
}
public PArray<outT> apply(PArray<inT> input) {
    try {
        return composition.apply(input);
    } catch (Exception e) {
        return deoptimize();  // Deoptimise to the multi-thread map
   }
}
```

Listing 4.7: Sketch of the `AcceleratorMap` operation

JPAI implements two versions of the *reduce* operation. JPAI implements a sequential reduction (`Reduce`) and a parallel reduction using Java threads `ReduceJavaThreads` as shown in Figure 4.1. In the case of the reduction there is no direct support for the GPU with JPAI because, as Steuwer et. al [Steu 15] has shown, it can be solved with a combination of map on GPU, in which each GPU thread process loop sequentially, and a final reduction on CPU.

4.2.5 Copy Operations

JPAI implements skeletons to perform copy operations between the host and the device through the Java classes `CopyToDevice` and `CopyToHost`. These classes perform the data transfer from the host to the device and vice-versa in their `apply` methods.

Furthermore, when running GPU applications from managed programming languages like Java, an additional operation is needed before copying the data to the GPU (or to any other OpenCL device). Java has its own object representation and OpenCL has its own representation of primitive and arrays data structures. Before sending the data from the host to the GPU, the data representation has to be transformed from Java to OpenCL (called *marshalling*) and vice-versa (called *unmarshalling*). JPAI implements two extra copy operations through the Java classes (`Marshal` and `Unmarshal`) to implement these transformations. Chapter 5 will explain, in detail, the `marshal` and

```
1 CopyToDevice<Float> copyIn = new CopyToDevice<>();
2 ArrayFunction<Float, Double> function = new OpenCLMap<>(x -> x * 2.0);
3 CopyToHost<Double> copyToHost = new CopyToHost<>();
4 // Composition
5 ArrayFunction<Float, Double> composition = copyIn.andThen(function).andThen(
     copyToHost);
```

Listing 4.8: Copy composition with map for OpenCL.

`unmarshal` and their impact in performance as well as a runtime technique to avoid the marshalling.

4.2.6 Identity Operation

Identity is a special operation in JPAI that returns the same function passed as argument. This is particularly useful for **zip** (which combines multiple input arrays in one array) and **copy** operations.

Listing 4.8 shows an example of the copy operation of simple function from types `Float` to `Double` to execute on the GPU. Line 1 creates a copy object from the host to the device (which is an operation that inherits from the identity function) and line 3 creates a copy from the device to the host. Types match exactly with the input and output of the main function to compute on the OpenCL device in Line 2. Then, line 5 composes all the operations in one single function. In the case of the copy operations, the identity allows to create a new function only with the input data type, because it returns the same function.

4.3 Array Function Composition

This section explains how the JPAI operations are composed and executed. One of the key aspects of JPAI is that it allows function composition. This is that, every operation in JPAI (e.g. `map, reduce, zip, copy`), returns a new `ArrayFunction` that can be used for invoking new operations. In this way, JPAI composes a pipeline of operations that are lazily computed once the user invokes the `apply` method.

JPAI provides a factory class (a class that contains static methods to build JPAI operations) to reduce the boilerplate code and facilitate the creation of the operations.

```java
ArrayFunction<Float, Double> f1 = AF.map(x -> x * 2.0);
```

Listing 4.9: Map method factory in AF factory class.

```java
// f1 is the same ArrayFunction declared in Listing 4.9
f1.map(y -> { /* more computation */ });
```

Listing 4.10: Example of Map composition in JPAI.

The factory is called `AF` and it is just a wrapper to the `ArrayFunction` class. Listing 4.9 shows an example using the factory class for the map operation. Line 1 creates a map function for multiplying the input array by 2.

Allowing composability of `ArrayFunction` When a new operation is created using the factory class, the invoked operation returns a new `ArrayFunction`, allowing to compose multiple operations. The composition is possible through a new type of construction when two or more operations are linked together through the Java class `ArrayFunctionComposition` presented in Figure 4.1. For example, the `f1` variable created in Listing 4.9 is an instance of `Map`. If a new map function is concatenated to the existing `f1` function, like the one listed in Figure 4.10, JPAI composes the new operation using the `ArrayFunctionComposition` Java class, which links the two operations and specifies an order of execution between them.

Listing 4.11 shows the corresponding `map` implementation in the `ArrayFunction` class. It creates a new `ArrayFunction` and links the new expression with the previous one in the compute pipeline. This `map` receives a new function and the `ArrayFunctionComposition` links the current function (**`this`**) with the new. The constructor of the `ArrayFunctionComposition` stores the two function and when the `apply` method is invoked, the two functions are executed as follows:

$$f1.map(f2).apply(input) = f2.apply(f1.apply(input))$$

The second function (`f2`) receives the data processed by the first function (`f1`).

```java
// Map implementation in ArrayFunction class
public <T> ArrayFunction<inT, T> map(Function<outT, T> function) {
        return new ArrayFunctionComposition<>(this, new MapArrayFunction<>(
           function));
    }
```

Listing 4.11: Map implementation in `ArrayFunction`.

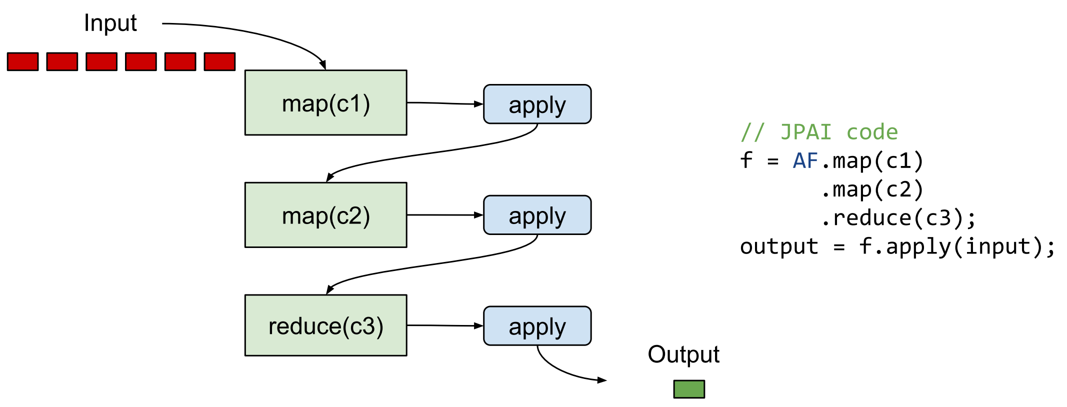
4.4 Array Function Execution

Figure 4.2 shows an overview of how parallel operations are executed in JPAI. It defines a compute pipeline with three operations (two `maps` and a reduction). When the computation pipeline is defined, JPAI creates a link for all the operations using the composition functions described in the previous section. When this pipeline of operations is executed, JPAI traverses and computes the `apply` methods for each operation defined in the compute pipeline.

Runtime Type Analysis When the compute pipeline is executed, it first need to allocate the output arrays. JPAI is implemented using Java generics. However, because of Java's type erasure, type information is not available at runtime. Java performs type erase at compilation time, and, therefore, at runtime there is no type information that can be used for performing the allocation of the corresponding `PArrays`. For this reason, JPAI performs runtime type analysis before executing each apply function to obtain the corresponding types.

Each apply method receives one input as a `PArray` and passes a new `PArray` to the next operation of the compute pipeline, as shown in Figure 4.2. The output `PArray` has to be instantiated before executing the next computation for each operation in the compute pipeline. To obtain the output types, JPAI executes on the CPU the function with only the first element of the input data set. Then, JPAI observes, at runtime, the output type and uses this information to allocate the corresponding `PArrays`.

This runtime type analysis process is a very simple strategy that allows executing the input function for a short period of time, giving the opportunity to the JVM to collect profiling information about the input program. This information is used when the code is compiled, at runtime, to the GPU via OpenCL.



```
// JPAI code
f = AF.map(c1)
      .map(c2)
      .reduce(c3);
output = f.apply(input);
```

Figure 4.2: `ArrayFunction` execution

Execution Once the runtime type analysis for the output allocation has been performed, JPAI executes the input computation passes to each parallel skeleton. Each parallel skeleton has its own implementation in the `apply` method. JPAI executes each operation and passes the result to the next operation. In the example shown in Figure 4.2 there are two maps and a reduction.

If the OpenCL skeleton is selected through the `OpenCLMap` or `MapAccelerator`, the execution step invokes the OpenCL backend. This OpenCL backend (so called Marawacc) obtains the lambda expression (input computation) and the input data from JPAI, generates and executes the code on the GPU.

4.5 Array Functions Reusability

Another important key feature of JPAI, in combination with composability already described in Section 4.3, is **reusability**. JPAI allows developers to reuse computation the same or different data multiple times. This is a key difference compared to other approaches such as Java 8 Streams, Apache Spark and Apache Flink, where the computation to perform is defined within the input data.

Listing 4.12 shows an example in Java that uses the Java Stream API. Line 1 creates an `ArrayList` and line 3 creates the stream. Line 4 computes a map function over the stream in parallel. Line 5 reuses the same stream to compute another map function. However, this application raises an exception at runtime, because the stream can not be reused once it has been consumed.

JPAI solves this issue by decoupling the data to the parallel operation. Listing 4.13

```java
ArrayList<Integer> data = new ArrayList<>();
initialiseList(data);
Stream<Integer> stream = data.stream();
Object[] array = stream.parallel().map(x -> x + 100).toArray();
array = stream.parallel().map(x -> x * 100).toArray();
System.out.println(Arrays.toString(array));
```

Listing 4.12: Illustration of reusability in Java 8 Stream.

```java
1  ArrayFunction<Double, Double> f1 = new Map<>(x -> x + 100);
2  ArrayFunction<Double, Double> f2 = new Map<>(x -> x * 100);
3  PArray<Double> input = new PArray<>(size);
4  initialiseInput(input);
5  PArray<Double> o1 = f1.apply(input);
6  PArray<Double> o2 = f1.andThen(f2).apply(input);
7  f1.apply(anotherInput);
```

Listing 4.13: Example of function reusability in JPAI.

shows an example in JPAI of two functions (f1 and f2) that are used to compute two different outputs in lines 5 and 6. Note that, in contrast to the Stream API, the f1 function can be reused multiple times to either concatenate to other operations (such as the one showed in line 6, or invoke with different input, as the one showed in line 7.

Reusability has also another important advantage for managed and JIT languages. If a method is executed multiple times, it is likely to be compiled to an efficient machine code. Therefore, if the lambda expression is reused, it is very likely that the lambda is compiled and optimised by the JIT compiler, and, therefore, increase the overall performance.

4.6 Benchmark Applications

The first part of this chapter has presented JPAI. This section presents how to use JPAI in real Java applications. It presents five different applications implemented with JPAI ranging from linear algebra (*saxpy*), to applications from the fields of mathematical finance (*Black-Scholes*), physics (*N-Body* simulation), computational statistics (*Monte*

```java
1 float alpha = 2.5f;
2 ArrayFunction<Tuple2<Float, Float>, Float> saxpy;
3 saxpy = AF.<Float, Float> zip2().map(v -> alpha * v._1() + v._2());
4 PArray<Float> result = saxpy.apply(left,right);
```

Listing 4.14: Saxpy implementation using JPAI.

Carlo simulation) and machine-learning (*k-means* clustering). The benchmarks were selected from well-known benchmark suites such as the AMD OpenCL SDK [SDK 16] and Rodinia [Che 09] and implemented in Java using JPAI.

Benchmarks selection JPAI exploits data parallelism via the array programming interface. The benchmarks were selected to follow this data parallel programming style. They are used in very different domains such as physics, video game engines and machine learning.

These benchmarks are compute-intensive applications and are normally implemented in low-level programming languages such as C, C++ or Fortran in order to obtain good performance. Using JPAI allows developers to execute on heterogeneous devices by offloading the input program to an accelerator at runtime from Java. This section discusses how these benchmark are implemented in Java by application developers using the pattern based JPAI.

4.6.1 Single-Precision alpha X Plus Y (*saxpy*)

The first application is a simple linear algebra benchmark which scales a vector with a constant α and then adds it to another vector. Listing 4.14 shows the implementation of *saxpy*. The two input vectors are combined into an array of pairs using the `zip` pattern in line 3. The following `map` pattern is then used to specify how to process each pair of elements. The computation is specified using a Java lambda expression (see line 3), where two corresponding elements of both vectors are added together after the first vector has been scaled with `alpha`. This example shows how ordinary variables defined outside of the patterns, such as `alpha`, can be accessed inside the lambda expression.

```
1 ArrayFunction<Float,Tuple2<Float,Float>> bs;
2 bs = AF.<Float> map( stockPrice -> {
3    // compute values
4    parameters = computation( stockPrice )
5    float call = call(parameters);
6    float put  = put(paratemers);
7    return new Tuple2<>(call, put);
8 });
9 PArray<Tuple2<Float, Float>> result = bs.apply(sPrices);
```

Listing 4.15: Sketch of the implementation of the Black-Scholes application using JPAI.

4.6.2 Black-Scholes

The Black-Scholes financial mathematics application is a standard method used is high frequency trading and it computes the two values, *call* and *put*, for stock prices. Listing 4.15 shows a sketch of the implementation in Java using JPAI. The algorithm was taken from AMD OpenCL SDK benchmark suite. This application receives a list of floating numbers and returns a list of Tuples for call and put stock prices.

The `map` pattern is used in line 2 to compute the *call* and *put* options for a given stock price. The input value is used to compute all the parameters needed to compute the call and put values. The `map` returns the call and put options encoded in a tuple (see line 7). Finally, in line 9 computation is applied to the array of stock prices.

4.6.3 N-Body Simulation

N-Body simulations are widely used in physics and astronomy to simulate the behaviour of a system of particles (a.k.a., *bodies*) over time. In an iterative process, forces between each combination of pairs of bodies are computed from which the velocity of each body is derived.

The N-Body JPAI implementation (shown in Listing 4.16) encodes the bodies as tuples with seven elements (represented as ... in the listing for simplification): the first three elements encode the position; the next three elements encode the velocity; and the last element encodes the mass of the body. N-Body computes a new array of bodies `newBody` (meaning a new position and its corresponding velocity for each new

```
1 ArrayFunction<Tuple7<...>, Tuple7<...>> nBody =  AF.zip7().map( body -> {
2     for (int i = 0; i < numBodies; i++) {
3        float[] acc = computeAcceleration(body, bodies[i]);
4     }
5     float[] f = updatePosition(body, acc);
6     float[] v = updateVelocity(body, acc);
7     Tuple7<...> newBody = updateBody(body, f, v);
8     return newBody;
9 });
```

Listing 4.16: N-Body simulation in JPAI.

body) by applying the corresponding computation. Inside the `map`, it first computes the force which acts upon body (`body`) from all other bodies before it passes the new position update its force and velocity (see lines 2–6).

4.6.4 Monte Carlo Simulation

Monte Carlo simulations perform approximations using statistical sampling, i.e., by generating and analysing a large number of random numbers. This technique is useful in applications ranging from risk management, physics simulation, to data analysis. In this *montecarlo*'s version Listing 4.17, a large number of random values are generated in order to approximate the value of π. Each random number is used as a seed to initialize the pseudo-random number generator on the GPU generating a large amount of random numbers.

4.6.5 K-means Clustering

K-means clustering is a method used in machine-learning to *cluster* similar data points. Its an iterative process containing two steps: 1) each point is assigned to its nearest cluster; 2) a new cluster center point is computed based on all cluster points.

Listing 4.18 shows a sketch of the implementation in Java using JPAI. It only expresses the first step using JPAI (distance computation). The computation is executed as an iterative process until a termination condition is met (see line 11). A data point is represented as a tuple of two float values, therefore, a `zip2` pattern is used together with the `map` pattern (see line 2) to compute the nearest new cluster id for each point `p`. Because this computational step requires access to the cluster centres, it defines the

```java
ArrayFunction<Float, Float> monteCarlo = AF.<Float> map( seed -> {
   for (int i=0; i<iter;++i) {
     seed = updateSeed(seed);
     float x = random(seed);
     float y = random(seed);
     if (dist(x, y) <= 1.0f) {
        sum++;
     }}
   return sum / iter;
   });
PArray<Float> result = monteCarlo.apply(seeds);
```

Listing 4.17: Monte Carlo simulation implemented with JPAI.

computation inside the iteration where the `centre` variables can be accessed inside the lambda expression.

4.7 Evaluation of JPAI for Multi-Core Systems

This section presents the experimental evaluation of JPAI using the CPU with Java threads. First it describes the experimental setup and then the performance evaluation for a multi-core CPU.

4.7.1 Experimental Setup

The benchmarks are evaluated in a four-core Intel i7 4770K processor with hyper-threading technology enabled. This processor is able to execute up to eight hardware threads concurrently. The hyper-threading has been kept enabled to mimic the typical computer system of current Java users. This system has 16GB of RAM.

The Java version used is Oracle JRE 8 (1.8.0.65). It is hard to do a fair comparison with Virtual Machines. VMs are complex pieces of software that perform many tasks at the same time, such as garbage collection, compilation and bytecode interpretation. In order to reduce the noise when measuring, the numbers reported in the comparison are peak performance. To do so, each benchmark has been executed 100 times and the median runtime is reported. Section 5.6.1 provides a detailed description about how this methodology was conducted to measure all the benchmarks for GPUs and

```java
do { // step 1
    m = AF.<Float,Float>zip2().map( p -> {
      // compute membership
      for (int i = 0; i < ks; i++) {
         float dist = distance(..)
         id = updateDist(dist); }
      return id;
    } ).apply(points);
   // step 2: update the points
   centers = update(points, m);
} while (condition);
```

Listing 4.18: Kmeans classification using JPAI.

Application	Description	Data Sizes		Domain
		Small	Large	
Saxpy	Vector Multiplication	2 * 8M	2 * 64M	Linear Algebra
K-means	Data classification	2 * 4M	2 * 32M	Clustering
Black-Scholes	Finance prediction	16M	64M	Mathematical Finance
N-body	Particle simulation	4 * 64K	4 * 512K	Physics
Monte-Carlo	Montecarlo simulation	256K	2M	Mathematical Finance

Table 4.1: Data size in MB for each benchmark used for the experimental evaluation.

CPUs (including the presented evaluation in this chapter). For this section, the baseline corresponds to the Java sequential version that is used to compare against parallel versions implemented using JPAI running on the multi-core CPU.

Table 4.1 shows the data sizes used for the experimental evaluation. The performance is measured using two input data sizes for each benchmark.

4.7.2 Runtime Results of Java Multithread Execution on CPU

This section evaluates all benchmarks to compare JPAI parallel framework against sequential Java implementations operating on arrays of primitive types. This is the way in which Java programmers implement performance oriented applications reaching performance close to native written C code.

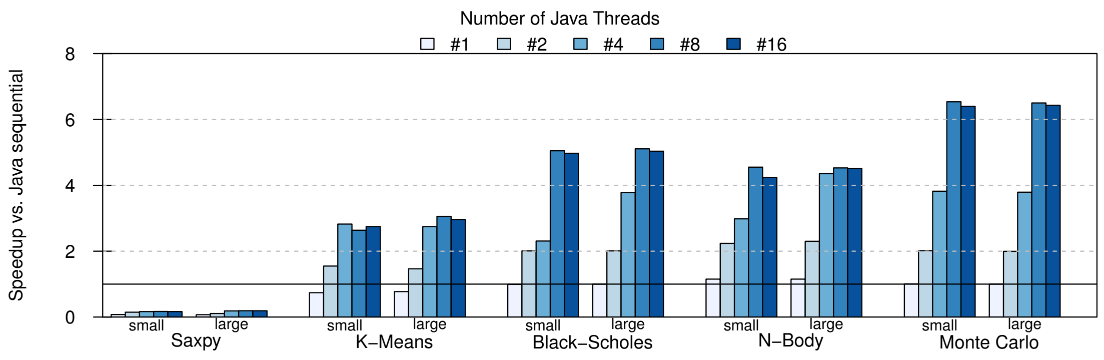


Figure 4.3: Speedup for all benchmarks over sequential Java with multiple threads.

The CPU has four cores with hyper-threading. All the benchmarks are implemented as explained in Section 4.6. Performance is reported using 1, 2, 4, 8, and 16 Java threads. The results are shown in Figure 4.3. The y-axis shows the speedup of the JPAI implementation over the sequential Java implementation.

The first benchmark *saxpy* is a memory intensive benchmark with few computations performed per memory operation. Therefore, no speedup is expected to see as all executing threads wait for the data to be read from memory. The sequential Java application already uses the optimal memory layout, a primitive `float` array. However when running JPAI , the `PArray` data structure is implemented using Java array buffers (as Chapter 5 will explain in detail). This introduces an overhead because of the use of `FloatBuffer` in combination with launching the Java threads.

The other four benchmarks show substantially better results, as these are benchmarks more suited for parallel execution. For the *K*-Means application, the runtime reported is the first step of the algorithm (see Section 4.6.5), where using JPAI gives a speedup of over 3× compared to sequential Java for the largest data size. For Black-Scholes, N-Body and Monte Carlo, speedup of up to 5× can be achieved in a multi-thread scenario. This is possible through the hyper-threading on the CPU, which increases performance up to 30% when this feature is enabled [Marr 02]. Hyper-threading interleaves the execution of multiple threads on the same physical CPU core, allowing a more efficient use of the CPU core resources.

The standard deviation compared to the median value (the reported value that is used to report peak performance) is between 0.6% and 21% for small data sizes and between 0.1% and 3% for large data sizes. For small data sizes, there is more variation since the profiler needs more iterations to compile the code.

The single threaded implementation using JPAI introduces no overhead, or only a moderate overhead, as compared to sequential Java for these five benchmarks. Overall, these results show that, JPAI is well-suited for implementing data-parallel applications in Java even when the power of only one multi-core CPU is used.

4.8 Summary

This chapter has presented the design and implementation of the JPAI, a Java Programming Array Interface, for parallel and heterogeneous programming within Java. JPAI allows Java developers to use the parallel resources of computers with minimal effort, focusing on algorithms rather than on the way they are implemented. JPAI implements a set of parallel skeletons using Java threads and OpenCL. JPAI is easily extensible for other parallel programming models and hardware technologies.

This chapter has also presented a detailed analysis of the programmability of five different scientific applications in Java with JPAI that are normally written in low-level programming languages, such as C or Fortran. Finally, an evaluation of those applications on a multi-core Intel CPU using Java threads has been presented.

Next chapter will show how JPAI is used to compile, at runtime, the user function to OpenCL and executed on a GPU by using Marawacc, a new OpenCL JIT compiler and a runtime system.

Chapter 5

Marawacc: A Framework for Heterogeneous Computing in Java

The previous chapter has presented an API designed for parallel and heterogeneous programming within Java. This API (JPAI) provides all the information needed for generating OpenCL code at runtime, such as the user computation, the input data, data types, scope variables and parallel skeleton.

This chapter presents Marawacc, a framework for heterogeneous computing in Java. The Marawacc framework is composed of an OpenCL JIT compiler that automatically translates Java bytecode into OpenCL C code, and a runtime system that orchestrates the GPU execution from Java. With Marawacc, Java applications can automatically be compiled to OpenCL at runtime and executed on a GPU.

Generating code for OpenCL from high-level programming languages like Java is not enough to get good performance on GPUs. In order to execute an OpenCL application from Java, the runtime has to convert the data representation from Java to the OpenCL's representation and vice-versa. This process is called marshalling. Marshalling is a very slow process that can easily degrade the overall performance when using GPUs. This chapter presents a technique to avoid the marshalling between Java and OpenCL, and therefore, speed-up the overall runtime of Java applications when using GPUs. This chapter also presents an evaluation of the Marawacc JIT compiler for OpenCL on a wide set of benchmarks and it compares Marawacc to state-of-the-art OpenCL frameworks for Java.

This chapter is organized as follows: Section 5.1 motivates this chapter. Section 5.2 shows the overall system design. Section 5.3 discusses how Marawacc generates OpenCL code from Java and executes the generated program on the GPU. Section 5.4

explains, in detail, how the OpenCL code is generated using the Graal intermediate representation. Section 5.5 introduces and discusses the runtime for data optimisation to avoid the marshalling process between Java and OpenCL. Section 5.6 shows a performance evaluation that compares Marawacc to Aparapi, JOCL and native OpenCL implementations. Finally, Section 5.7 summarizes this chapter.

5.1 Motivation

The previous chapter has presented the Java Parallel Array Interface (JPAI) for parallel and heterogeneous programming as well as a set of applications and their performance in a conventional CPU. Programs written with JPAI can execute a Java application by using Java threads or OpenCL. Now the question is how to execute, from JPAI, Java applications on GPUs automatically.

The primary goal of this thesis is to simplify the usability and programmability of heterogeneous devices from interpreted languages like Java. JPAI simplifies programmability to end users, but there must be a runtime system that transforms the input application into valid GPU code. This is performed via a JIT compiler that automatically translates the input Java expressions when using the JPAI programming interface into OpenCL. This chapter shows a system which can compile, optimise and execute a Java application into the GPU at runtime automatically.

There are several Java projects that generate OpenCL C code, at runtime, from Java bytecode. The most mature projects are *AMD Aparapi* and *RootBeer*. These projects use an API for programming heterogeneous systems and a compiler that translates Java bytecode into OpenCL C. In the case of AMD Aparapi, developers extend a Java class and override a kernel method that will be executed on the GPU. In that method, programmers specify the OpenCL thread-id by calling the `getGlobalID()` method available in Aparapi. Although the computation is expressed in pure Java, this methodology remains low-level for non-expert programmers, because it is very similar to how OpenCL kernels are implemented. Besides, AMD Aparapi does not support Java number wrappers such as `Integer`, classes such as `Tuples` and static array allocation. The set of compiler optimisations that AMD Aparapi applies, are not integrated with the standard JVM. For example, hardware independent optimisations such as loop unrolling, constant propagation, escape analysis or method inlining are not reused from the JVM compilation pipeline.

Rootbeer provides a similar programming interface than Aparapi, in which programmers also extends a GPU class and override a kernel method. However the GPU compilation is not at runtime. Alternately, the GPU code is generated statically by using an external tool.

This chapter presents Marawacc, a new approach to extend the Graal compilation pipeline to compile, at runtime, Java bytecode to OpenCL. Marawacc is composed of a JIT compiler and a runtime system that executes and optimises Java applications on GPUs. It is focused on data parallellism and uses the map parallel skeleton defined in JPAI. Marawacc is based on the Graal compiler and it generates OpenCL C code using the Graal Intermediate Representation (Graal IR). Marawacc reuses and takes advantage of the existing Graal compiler optimisations where the OpenCL JIT compilation is invoked as a new phase in combination with the existing ones in Graal.

Marawacc also uses information from JPAI, presented in Chapter 4, such as the parallel skeleton, input and output data, types and the user function. When performing heterogeneous computing from Java, data management is a very important aspect with a direct impact on performance. Prior to perform the data transfers, an explicit transformation of data representation between Java and OpenCL C is required. This type of transformation between programming languages is called marshalling. As this chapter shows, the marshalling process is very time consuming. This chapter addresses the issue of data transformation and presents a technique to avoid the marshalling and unmarshalling transparently.

This chapter makes the following contributions:

- It presents an OpenCL JIT compiler that transforms Java bytecode to OpenCL C code at runtime. The OpenCL JIT compiler is integrated into the Graal VM and it can be used as a new compiler phase in combination with the existing ones in Graal.

- It presents an optimising runtime that is able to execute OpenCL applications on GPUs from Java. The runtime also optimises the data transformations between Java and OpenCL, allowing to increase the overall performance.

- It presents an evaluation where the Marawacc framework is compared to Aparapi, JOCL and native OpenCL. It shows that Marawacc is, on average, 55% faster than Aparapi and 23% slower than native implementations.

5.2 Marawacc System Overview

Marawacc is designed to allow programmers to use available parallel accelerators (e.g. GPUs) transparently from the Java language. It is built around two major components: (1) an OpenCL code generator and (2) a runtime that manages and optimises the execution of the computation on OpenCL devices.

Marawacc makes use of JPAI to obtain the compute function, the input data and the parallel skeleton to be generated. The previous chapter explained the design and implementation of JPAI. This chapter is focused on the OpenCL runtime code generation and the efficient runtime data management between Java and OpenCL. The OpenCL code generator in Marawacc extends the Oracle Graal VM [Dubo 13b], a novel JIT compiler written in Java, to compile Java bytecode to OpenCL C. The second component orchestrates the OpenCL generated binary from Java to execute the user program on the GPU. It also manages GPU code caches, performs data transfer optimisations between the host and devices and optimises the marshalling process between Java and OpenCL.

The Marawacc code generator uses an API provided by Graal for processing standard Java bytecode and uses it to generate OpenCL code. Marawacc, therefore, does not require the Graal VM to compile and execute Java applications on the GPU, but uses functionality and utilities provided by the Graal API which facilitates the compilation and runtime code generation.

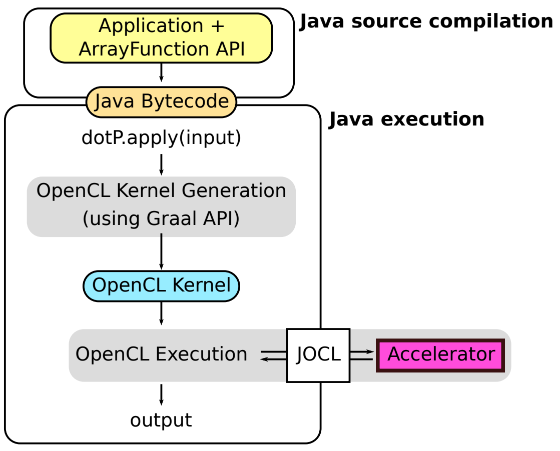


Figure 5.1: Overview of Marawacc.

Figure 5.1 shows an overview of Marawacc and how the runtime performs execution on an accelerator. The user application is completely written in Java, in which the parallelism is expressed with the help of the Java array programming interface (JPAI) using operations such as `map` and `reduce`. The Java application is then compiled into Java bytecode that is executed in a standard JVM. At runtime, when the User Defined Function (UDF) is first executed with JPAI, the Marawacc runtime checks the system configuration and determines whether it should attempt to execute the computation on an OpenCL device (e.g. on a GPU) or not. If there is no

OpenCL device available, Marawacc falls back to a Java implementation that uses Java threads. If there is an OpenCL device available, the Marawacc runtime attempts to accelerate the computation by using that device. To generate the OpenCL C code, the Graal API is used to build the Graal intermediate representation (Graal IR). Then, the Marawacc runtime executes the GPU code on the accelerator using the JOCL OpenCL Java binding [JOCL 17], as it will be described in Section 5.5. If there is a runtime error (due to an unsupported feature or an OpenCL runtime error) the execution aborts and the computation resumes safely in Java.

In contrast with prior works such as LiquidMetal [Duba 12], Marawacc generates the OpenCL kernel at runtime, once it detects a usable OpenCL device. With Marawacc, user applications are packaged as standard Java bytecode without any device specific code. This is an important feature since Marawacc keeps the philosophy of *compile-once run-everywhere*. In addition, this offers an opportunity for future work to specialise the code for the available device. The same design can be used to support another device-specific back-end (e.g. CUDA PTX) or to implement device specific optimisations.

5.3 The Marawacc System: Compilation and Execution

This section describes the Marawacc OpenCL JIT compiler that translates Java bytecode to OpenCL C code. This allows programmers to execute computations seamlessly on GPUs. Marawacc extends the Oracle's Graal compiler [Dubo 13b] for OpenCL optimisations and code generation.

The OpenCL compilation process in Marawacc is shown in Figure 5.2. With Marawacc, programmers write the Java applications using the JPAI programming interface described in Chapter 4.2. Then, the Java application is compiled to Java bytecode using the `javac` compiler. Note that there is no modification of the Java language. The Java bytecode contains a call to the `apply` method of the lambda expression used in the map primitive in JPAI, as shown on the left side of Figure 5.2. The Marawacc runtime inspects the `apply` method by using reflection and obtains the compute functions passed as argument to the map parallel primitive.

The optimisation process happens at runtime, inside the Graal VM. When the map parallel skeleton is executed, the Marawacc OpenCL JIT compiler transforms the original Java bytecode to OpenCL C code performing four steps:

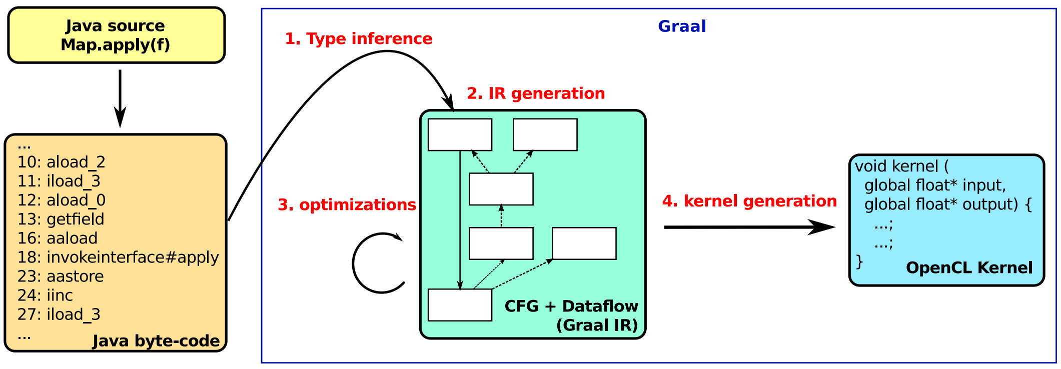


Figure 5.2: Marawacc OpenCL code generation process from the Java bytecode to OpenCL C. 1) Marawacc performs type inferences for the input and output. 2) The bytecode is converted into the Graal IR and 3) various optimisations are performed. 4) Finally, the Marawacc backend generates an OpenCL kernel.

1. Marawacc performs runtime data analysis to obtain the data types of input and output variables as well as scope passed to the lambda expression. JPAI, as described in Chapter 4, uses Java generics to assure the data types safety. However, when compiling the application using `javac` the data types are erased, and, therefore, they have to be inferred again by Marawacc.

2. The Java bytecode is converted into the Graal intermediate representation in the form of a control flow and a data flow graphs. Marawacc invokes the Graal API to parse the bytecode and forms the CFG.

3. Marawacc optimises the CFG using existing Graal compiler phases and new compiler phases specilised for GPU code generation. Some example of optimimisation compiler-phases are inlining and node replacement.

4. Marawacc generates the OpenCL C code by traversing the CFG and generating the corresponding OpenCL source code for each node in the CFG.

Once the OpenCL kernel is generated, Marawacc inserts the source code into a cache and reuse it if the user calls to the same function in the Java program again. Therefore, the OpenCL code generation is only performed the first time the `apply` method is invoked. Then, the Marawacc runtime executes the OpenCL program on GPU. The following sections describe and discuss in detail each step in the compilation pipeline from the type inference to the final OpenCL code generation.

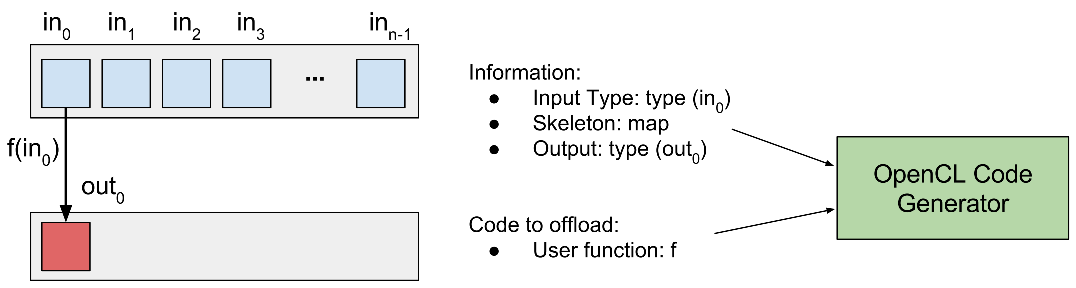


Figure 5.3: Runtime type analysis in Marawacc before generating the OpenCL kernel.

5.3.1 Runtime Type Analysis

To generate an OpenCL C kernel from Java, Marawacc requires the user defined function and the parallel skeleton plus some additional information. Figure 5.3 shows all the information required to compile to OpenCL C.

Marawacc obtains most of the information from the JPAI array implementation, which corresponds to the input and output data types, the scope variables and their types, the `map` parallel skeleton and the user function. All of this information is used to transform the Java bytecode to OpenCL C code.

However, due to the fact that JPAI interface is implemented using Java generics, the type information is erased by the Java compiler `javac`. In order to obtain the types, Marawacc has to perform runtime type analysis for the output and scope variables. Figure 5.3 shows how Marawacc performs the runtime type analysis to obtain the output data type. Marawacc executes the input function on the CPU sequentially with the first element from the input data set. In the case of the map pattern, Marawacc simply runs the lambda expression on the first element of the input array. Then, by using Java runtime reflection, Marawacc obtains the output type of each expression. All of this information is stored internally and it is handled by the Marawacc compiler framework to compile to OpenCL in later phases.

JPAI allows programmers to use variables that are defined outside the scope of the compute method inside of the lambda expressions (so called lexical scope variables). The JPAI implementation provides a set of utility methods to automatically detect and capture the lexical scope variables. The JPAI implementation makes use of Java reflection to obtain the data and their types. This information is then stored internally and passed to the code generator, which generates the corresponding types, and to the runtime, which allocates the device buffers and transfers the data.

```java
public <T, R> void mapSkeleton(T[] input, R[] output) {
    for (int i = 0; i < input.length; i++) {
        T aux = input[i];
        output[i] = (R) function.apply(aux);
    }
}
```

Listing 5.1: Map parallel skeleton used for building the CFG in the Graal IR form

5.3.2 Graph Builder: Java Bytecode to Graal CFG

Once Marawacc performs the runtime type analysis and collects all the information required for the OpenCL code generation, it transforms the Java bytecode that corresponds to the input user function (or lambda expression) to a CFG in the Graal IR form. In fact, Marawacc builds two CFGs: one for the input function and another one for the parallel skeleton.

The CFG for the input function corresponds to the lambda expression passed as argument to the map parallel skeleton. Marawacc invokes the Graal compiler to parse the Java bytecode that represents the lambda expression into a CFG in the Graal IR form. This graph is used in later compiler phases to optimise and generate the OpenCL C code.

Marawacc also builds a CFG for the parallel skeleton that is selected in the API (JPAI). For example, if a map operation is invoked in JPAI, then the Marawacc runtime builds the CFG for the map parallel skeleton. The CFG for the parallel skeleton is built from a specific Java method used as a skeleton for the code generator. The Java code for the map skeleton is shown in Listing 5.1. This skeleton contains the sequential implementation of the pattern in pure Java. During code generation, the Marawacc code generator translates the whole method to the main OpenCL kernel function from the Graal IR. At the end of this process (graph builder), Marawacc has two CFGs in memory: one for the user function and another one for the parallel skeleton. These two graphs are inlined, optimised and used by the OpenCL code generator in later phases of the compilation pipeline.

5.3.3 GPU Optimisation Phases in the Graal IR

After building the CFG using the Graal API, Marawacc performs a set of compiler phases that optimises the graph before the OpenCL code generation. The OpenCL optimisations are implemented as a new set of compiler-phases used in combination with the Graal compilation pipeline for CPUs.

In this manner, the OpenCL optimisation phase can be invoked in addition to the Graal optimisations. This is a key distinction with the related work projects, such as Aparapi, Rootbeer and IBM J9, where common compiler independent architecture optimisations are not applied. The Marawacc OpenCL compiler works together with the Graal compiler to take advantage of the common and powerful Graal optimisations already implemented, such as inlining, constant propagation, constant folding, etc.

Compiler Transformations All compiler transformations operate on the CFG, by removing existing nodes or adding new nodes. Marawacc starts the optimisation process by applying architecture-independent compiler optimisations to the control flow graphs:

- Canonicalizing: local optimisations, e.g., constant folding, constant propagation, local strength reduction (where numeric expressions are replaced by less expensive operations [Coop 01]).
- Inlining: method calls are replaced by their implementations.
- Dead code elimination: code segments never executed are removed from the graph.
- Partial escape analysis: the compiler determines whether the object escapes or not the actual condition inside a method. If it does not escape the scope, the object can be allocated on the stack instead of the Java heap, allowing the VM to reduce the amount of work performed by the GC. This optimisation has a huge impact in overall performance for Java applications [Stad 14b].

All of these hardware independent optimisations exist already in Graal. The OpenCL back-end compiler uses these optimisations before generating the corresponding OpenCL code. These optimisations are important to support Java constructs that would not be possible if there would have been a direct translation between Java bytecode and OpenCL C. An example is the Java array allocation with the `new` keyword. In other

```java
ArrayFunction<Integer,Double>
functionScalarVector = AF.map<>(x -> (double)x*2; );
```

Listing 5.2: Java example using Marawacc API for GPU programming

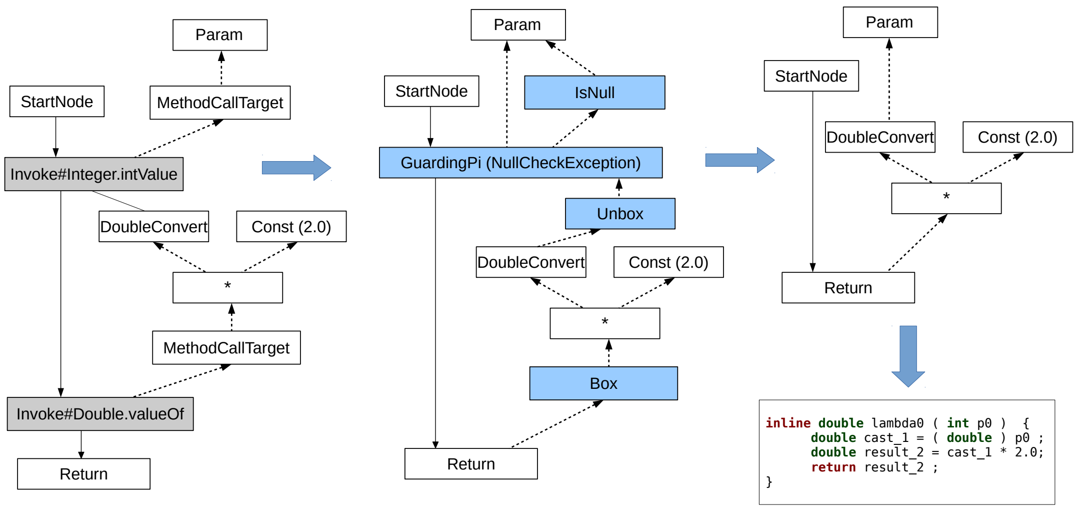
projects such as Aparapi, static arrays can not even be allocated. With Graal, optimisations such as constant propagation, value numbering and strength reduction can be applied to provide the concrete size that will be used to allocate the static array. Therefore, Marawacc will generate a static array in the OpenCL kernel with the corresponding size.

OpenCL Compiler Transformations Marawacc also applies additional compiler transformations to remove and transform IR nodes, which are not required in an OpenCL kernel or have corresponding implementation in OpenCL. For example, `null` check references are removed from the input and output arguments since it can guarantee that these are not `null` outside of the kernel. It eliminates the (un)box nodes by converting all wrapper objects, such as `Float` or `Integer`, to their corresponding primitives equivalent (`float`, `int`). For example, method calls to the Java math library, which invoke to native code (e.g. `Math.cos`), are transparently replaced with new calls to equivalent OpenCL built-in math functions.

An example of these compiler transformations can be seen in Figure 5.4. The graph corresponds to the CFG for the input Java expression in Listing 5.2 where dash arrows represent data flow and solid arrows represent control flow. The `map` is a function from `Integer` to `Double` data types.

When parsing Java bytecode into the Graal IR, Marawacc generates the graph on the left side of Figure 5.4. Each graph has an entry point represented as a `StartNode` in the IR. The parameter node (`Param`) at the top corresponds to the input parameter. The highlighted nodes (`Invoke`) correspond to the calls to get the value of the input data from the Integer class. The call target node (`MethodCallTarget`) in the IR together with the `Invoke#Integer.intValue` node invoke the unboxing of the `Integer` object into an integer primitive.

After applying the standard Graal inlining and dead code optimisation passes, Marawacc obtains the graph in the middle of Figure 5.4, where the `invoke` nodes representing method calls have been replaced by `IsNull` and `Boxing/UnBoxing` nodes. The `Invoke#Integer.intValue` nodes is lowered to an `Unbox` and a null pointer check nodes.



```
inline double lambda0 ( int p0 )  {
      double cast_1 = ( double ) p0 ;
      double result_2 = cast_1 * 2.0;
      return result_2 ;
}
```

Figure 5.4: Graal IR optimisation phases and OpenCL C code generation. The left side shows the Graal IR for the Java input code. Then, the inlining target optimisation is invoked producing the middle graph. As Marawacc produces OpenCL C code, it does not need to check for `null` pointers and perform (un)boxing, therefore, the graph is optimised to the the rightmost graph, from which the shown OpenCL C code is generated.

Similarly the `Invoke#Double.valueOf` node is lowered into a `Box` node. This means that there are new checks that ensure that the input data is not **`null`**. If that is the case, there is a call to fall back (`GuardPiNode`). However, when running on GPUs, the Marawacc runtime has to perform a copy from the host to the device. This means that there is an extra check that ensures that there are not null references associated to the input data. Otherwise, Marawacc can not execute the code on the GPU, because it is not valid data.

Therefore, Marawacc can safely remove these redundant nodes, producing the graph on the right side of Figure 5.4. This optimised graph is then sent to the OpenCL C code generator.

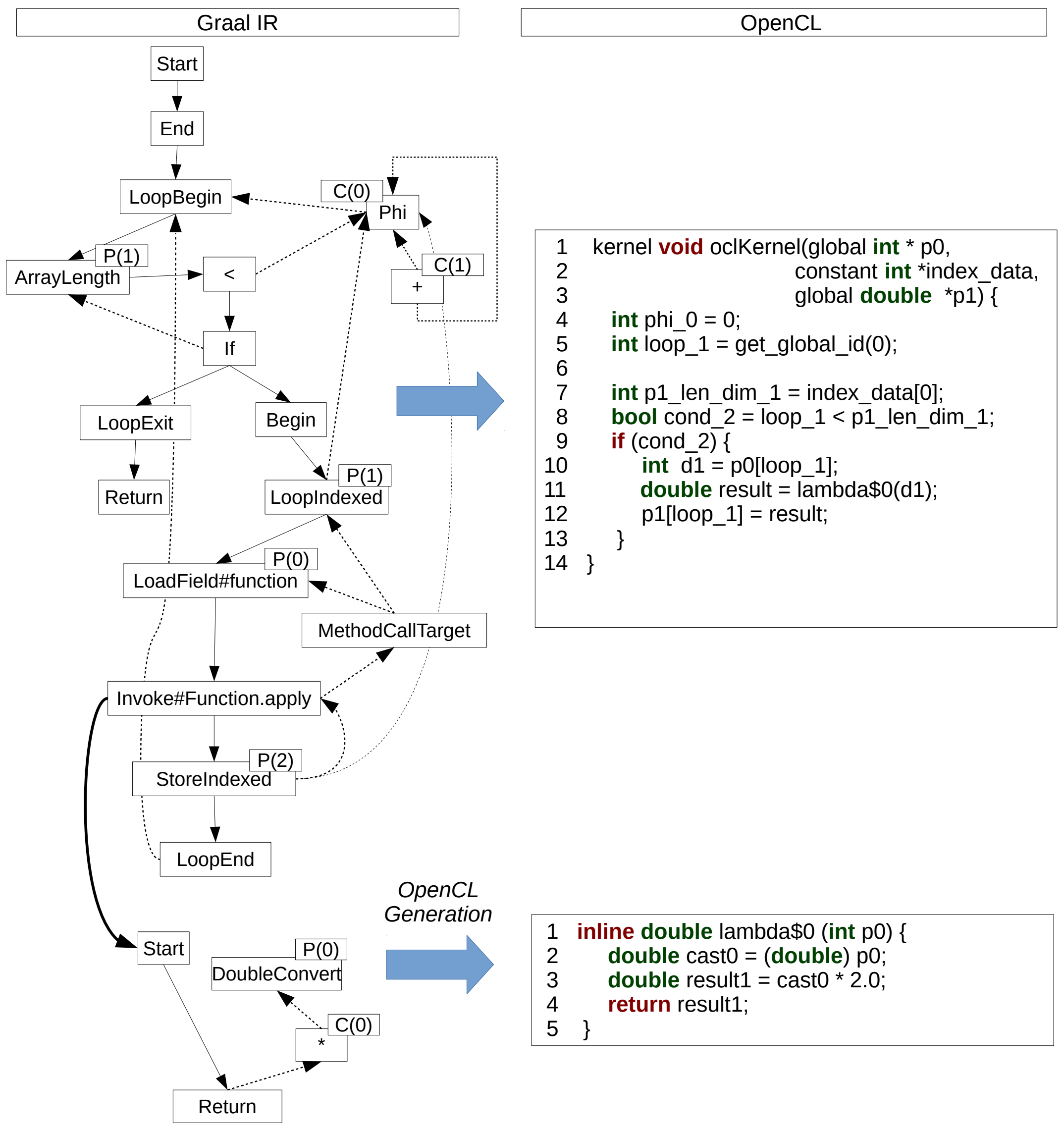
Handling Runtime Errors If at any point during the compiler transformation process Marawacc encounters a problem (e.g. a feature used in the lambda not supported in OpenCL), it aborts the OpenCL kernel generation process and falls back to a pure Java execution. In this manner, the Java code is always executed, that currently can be either on the GPU using OpenCL on a multi-core CPU using Java threads.

5.3.4 OpenCL C Code Generation from the Graal IR

This section presents the OpenCL code generation from the Graal IR. Consider the same Java program for scalar-vector multiplication shown in Listing 5.2. After applying the compiler transformations presented in Section 5.3.3, Marawacc sends the CFG to the OpenCL code generator. The input of the code generator corresponds with two CFGs, one for the parallel skeleton (such as `map`) and another one for the user computation. Marawacc keeps these two separate graphs to make the code generation easier.

Parallel Skeleton Figure 5.5 shows the corresponding graphs in the Graal IR form for the `map` parallel skeleton and the lambda expressions and its OpenCL code in the right side. The graph at the top left corresponds to the parallel map skeleton and its corresponding OpenCL C code is shown at the top right of the Figure. Marawacc first passes the CFG that represents the map parallel skeleton, to the code generator that emits the entry point for OpenCL (the main function of the OpenCL kernel). The black arrows represent control flow graph and the dash arrows represent data flow. `LoopBegin` node represents the entry point to the loop for the map skeleton. The Java for loop is mapped to a function call in OpenCL that obtains the thread-id. This corresponds to the function `get_global_id(0)` in line 5. Nodes `Invoke#Function.apply` and `MethodCallTarget` correspond to the method invocation of the apply method in JPAI. These nodes depend on the input data that is loaded from the first parameter (P0) of the lambda expression through the `LoadIndexed` node. These nodes are mapped to the line 11 shown in the OpenCL code on the right side.

Lambda Function Similarly to the OpenCL code generation of the parallel skeleton, Marawacc generates the OpenCL C code for the lambda expression. The CFG that representes the skeleton contains a call to the lambda expression (`Invoke#Function.apply`). The bottom left of Figure 5.5 shows the CFG for the lambda expression used in the Listing 5.2 after applying the optimisations described in Section 5.3.3. The bottom right part of the Figure shows the corresponding OpenCL C code. As Figure shows, the OpenCL C code is very similar to the input Java code shown in Listing 5.2 with an additional operation for type-casting in OpenCL from types **`int`** to **`double`**. Section 5.4 will show, in detail, how the translation between the Graal IR nodes and OpenCL is performed.



```
kernel void oclKernel(global int * p0,
                            constant int *index_data,
                            global double  *p1) {
   int phi_0 = 0;
   int loop_1 = get_global_id(0);

   int p1_len_dim_1 = index_data[0];
   bool cond_2 = loop_1 < p1_len_dim_1;
   if (cond_2) {
       int  d1 = p0[loop_1];
       double result = lambda$0(d1);
       p1[loop_1] = result;
   }
}
```

```
inline double lambda$0 (int p0) {
    double cast0 = (double) p0;
    double result1 = cast0 * 2.0;
    return result1;
}
```

Figure 5.5: Control Flow Graphs for the parallel skeleton and the lambda expression and their corresponding generated OpenCL C code.

5.4 Graal IR to OpenCL C

Marawacc uses a visitor pattern to translate the code from the Graal IR form to OpenCL C code. The OpenCL code generator contains a method per node in the IR plus a set of utility methods to generate the code. This section presents, in detail, how the OpenCL code generation is performed from the Graal IR.

```
Input: GraalIR $irNode$
dependencies = irNode.getDependencies();
for $Node \in dependencies$ do
    if $Node \notin symbolTable$ then
        dispatch(Node);
        addSymbolTable(Node);
    else

    end
end
emitOpeNCLCode(irNode, dependencies);
for $Node \in irNode.getSuccessors()$ do
    dispatch(Node);
end
```

Algorithm 1: General algorithm for generating OpenCL C code from the Graal IR.

Algorithm's Overview of the OpenCL Code Generation

Marawacc code generator makes use of a symbol table to create and assign variable names to data flow nodes in the Graal IR, such as expressions, constants, arrays, etc. Marawacc code generator constantly looks up in this symbol table to solve the data dependencies required for each node in the Graal IR.

Algorithm 1 shows the general procedure to generate OpenCL C code from Graal IR nodes. First, Marawacc obtains a list of data dependency nodes. This list will vary depending on the type of IR node. Then, if the dependencies are not in the symbol table, the code generator processes the dependencies and adds them to the symbol table. Once the dependencies are solved, the OpenCL C code can be emitted. Finally, the code generator continues exploring the successors of the current node in the control flow.

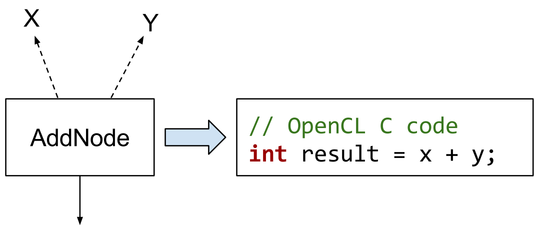
Types of IR nodes The Graal IR has two types of nodes: fixed nodes and floating nodes. Fixed nodes are connected with a list of predecessors and successors. They form the main shape of the CFG. Floating nodes are connected to the CFG through a list of input nodes. Floating nodes can be moved to any point of the CFG, allowing optimisations such as global value numbering and strength length reduction. CFGs in Graal have a start node (`StartNode`) that indicates the beginning of the method to be compiled. The OpenCL code generator starts to generate code from that node, which invokes the visitor that traverses each input node to generate the corresponding OpenCL C code. This section shows different nodes in the Graal IR and how they are mapped to OpenCL C code. For all the nodes shown in this chapter, dash lines represent data flow and solid lines represent control flow.

Generating OpenCL Code from the Graal IR

Marawacc supports 85 nodes from the high-level Graal IR between data flow and control flow nodes. The rest of this section shows, in detail, how these nodes are mapped to OpenCL C code.

Arithmetic Nodes All arithmetic nodes have the same IR form. They receive two inputs and produce one output. These are the nodes `AddNode`, `SubMode`, `MulNode`, `DivNode` and their specialized versions for integer and float as well as module operations.

As as example, Figure 5.6 shows the `IntegerAddNode` on the left and the corresponding OpenCL C code on the right. Marawacc first solves the data dependencies to obtain `x` and `y` expressions. Then, it produces the OpenCL C code for the addition and it adds to the symbol table the new variable (`result`) in order to use this variable in later nodes.

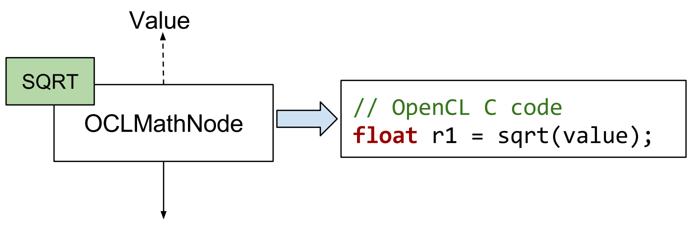


```
// OpenCL C code
int result = x + y;
```

Figure 5.6: OpenCL code generated for `AddNode`

OpenCL Math Node Marawacc performs node replacement for operations from the Java math library. It replaces operations such as `Math.sqrt` for a special node called `OCLMath`. This node contains a Java enumerate with the operation to compute. When Marawacc emits code for an `OCLMathNode`, it first inspects the operation required and then it emits the corresponding OpenCL C built-in function as shown in Figure 5.7.

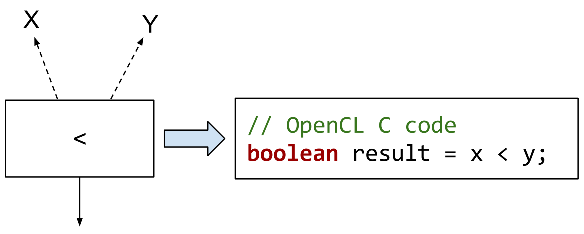
The reason to do this is that the Java math functions in Graal calls native code, which is not accessible from the OpenCL device. Instead, it emits the corresponding OpenCL C built-in.



```
// OpenCL C code
float r1 = sqrt(value);
```

Figure 5.7: `OCLMathNode` in the Graal IR and the generated OpenCL C code for `sqrt` operation.

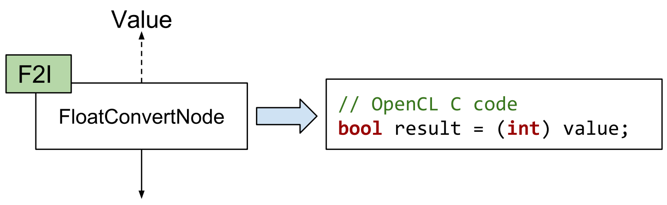
Logic Nodes Logic operations in OpenCL are generated in similar way to arithmetic nodes. Logic Graal IR nodes are `IntegerLessThanNode`, `FloatLessThanNode`, `IntegerEqualsNode` , `FloatEqualsNode` and `IntegerBelowNode`. Figure 5.8 shows an example where two integers are compared using the IR node `IntegerLessThanNode`.



```
// OpenCL C code
boolean result = x < y;
```

Figure 5.8: `IntegerLessThanNode` in the Graal IR and the generated OpenCL C code.

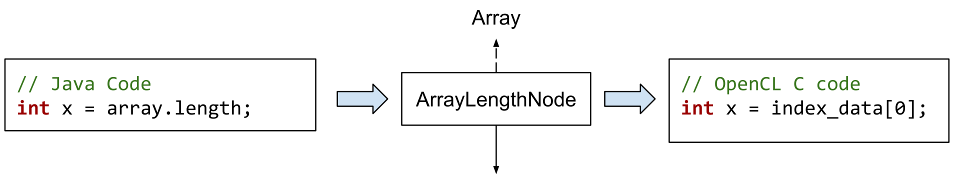
Conversion nodes In Graal, type conversion is represented through a node called `FloatConversion`. This node contains a tag with the type of conversion to be performed. Marawacc code generation inspects the enumerate and emits the corresponding type casting over the current node. Figure 5.9 shows an example of type conversion type **`float`** to **`int`**, following its tag `F2I`.



```
// OpenCL C code
bool result = (int) value;
```

Figure 5.9: `FloatConversionNode` in the Graal IR and the generated OpenCL C code.

Array Length Node The `ArrayLengthNode` in the Graal IR simple obtains the length of an array. The Graal IR node, represented in the graph shown in Figure 5.10, shows that the node receives an array as a dependency. During the OpenCL code generation, Marawacc does not pass the input data and arrays to the code generator. Instead, it generates a meta-data array for each input array. The meta-data arrays stores the array sizes. When the `ArrayLengthNode` is required, Marawacc generates an array access to the corresponding meta-data array to obtain the size in the OpenCL kernel. Figure 5.10 shows an example in Java that assigns the length of an array to a variable (on the left side), the corresponding IR node in the middle and its OpenCL code on the right side.



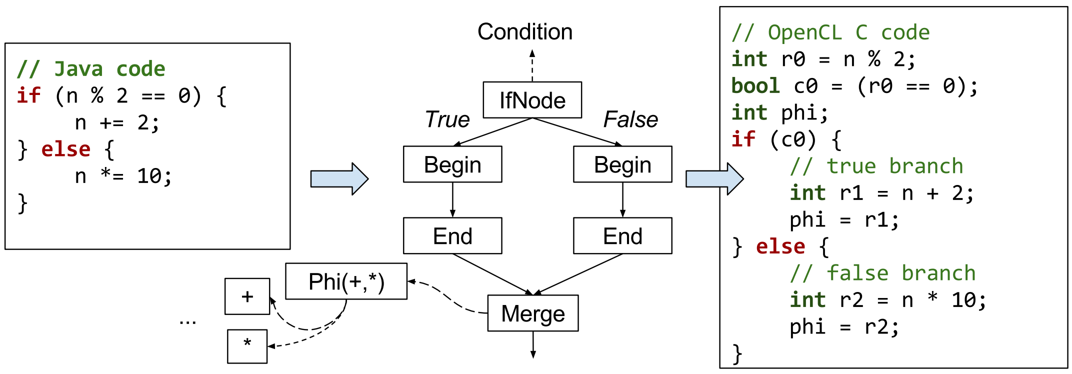
```
// Java Code
int x = array.length;
```

```
// OpenCL C code
int x = index_data[0];
```

Figure 5.10: `ArrayLengthNode` in the Graal IR and the generated OpenCL C code.

If Node As Section 2.4.3 has explained, Graal is represented in SSA (Static Single Assignment). This means that each variable is declared and used once. If the code has branches and a certain variable needs to be updated depending on the condition, the new variable has to be merged between two or more variables (modeled by so called `phi` nodes). Graal introduces `Phi` nodes and `Merge` nodes to merge data and control flow.

Marawacc first manages the `phi` variables by inserting all its nodes in a symbol table before generating the OpenCL C code. When the OpenCL code generator starts to emit the code, it first emits all the `phi` variables at the beginning of the block (e.g. at the beginning of the kernel function). When Marawacc traverses the IR to generate code and finds a *phi* dependency, the code generator updates its value in the corresponding branch.



```
// Java code
if (n % 2 == 0) {
    n += 2;
} else {
    n *= 10;
}
```

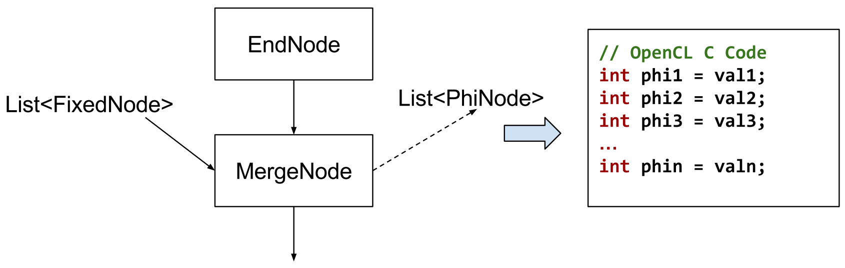
```
// OpenCL C code
int r0 = n % 2;
bool c0 = (r0 == 0);
int phi;
if (c0) {
    // true branch
    int r1 = n + 2;
    phi = r1;
} else {
    // false branch
    int r2 = n * 10;
    phi = r2;
}
```

Figure 5.11: `IfNode` in the Graal IR and the generated OpenCL C code.

The `IFNode` node in the Graal IR has one input representing the condition and two successors: the true and the false blocks. Figure 5.11 shows an example of an *if-else* condition. The left side of the figure shows an example of an `if` condition in Java, the middle graph shows the corresponding Graal IR and the right side shows the generated OpenCL C code.

To generate the OpenCL C code, Marawacc first solves the data dependency for the condition of the if node. Then it emits the string ¨`if (condition){`¨, and it visits the nodes for the true condition to emit the corresponding OpenCL code. If the false branch is present, then Marawacc also visits and emits the OpenCL code for the false condition. The `phi` variable is updated at the end of each block (true and false branches) with the corresponding value.

Merge and End Block Nodes As shown in Figure 5.11, **merge** nodes combine nodes from multiple control flow paths. An `EndNode` closes a block that was previously opened with `BeginNode`. An `EndNode` is always followed by a `MergeNode` as Figure 5.12 shows.



```
// OpenCL C Code
int phi1 = val1;
int phi2 = val2;
int phi3 = val3;
...
int phin = valn;
```

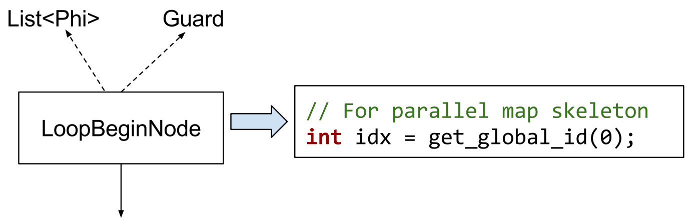
Figure 5.12: `MergeNode` in the Graal IR and the generated OpenCL C code.

When the code generator reaches a Graal `EndNode`, it inspects the corresponding `phi` values by using the immediate successor `MergeNode` and by updating the new phi-value. This strategy is also used not only for *if-else* nodes, but also for switches and loops.

For Loop Node The OpenCL code generation for the Graal `LoopNodeBegin` is handled specially. Marawacc generates the outermost loop using the OpenCL thread-id, which uses the OpenCL identifier of each thread to map the iteration index space. For inner loops, Marawacc generates sequential `for-loops`.

Figure 5.13 shows an example of generated OpenCL C code for the outermost loop. The IR node is `LoopBeginNode` a list of phi nodes and an optional input (guard) that controls if there is an overflow.

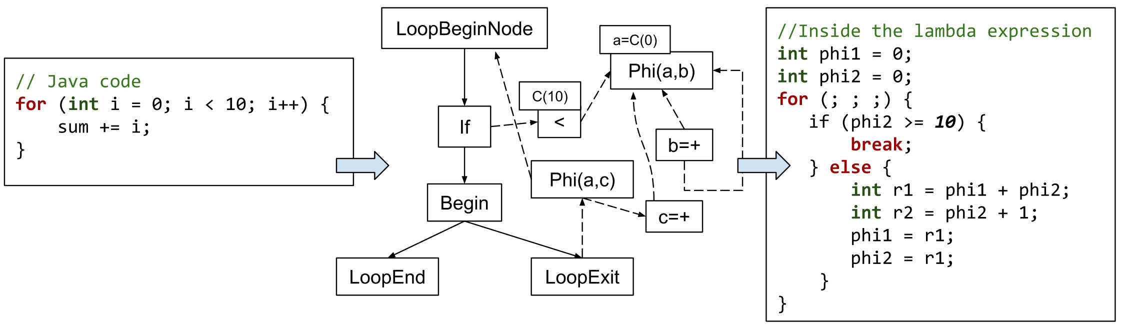
Figure 5.14 shows an example of generated OpenCL C code for sequential loops. Marawacc currently does not generate 2D or 3D kernels. To do that, it is necessary



```
// For parallel map skeleton
int idx = get_global_id(0);
```

Figure 5.13: `LoopBeginNode` in the Graal IR and the generated OpenCL C code for indexing the outermost .

that JPAI allows programmers the use of nested parallel skeletons (e.g. a map inside a map skeleton) in their applications. Therefore, if an inner loop has to be generated, Marawacc generates the corresponding sequential loop. The IR in the Figure shows the two `phi` nodes that are used in the OpenCL code to index and compute the `sum` value.

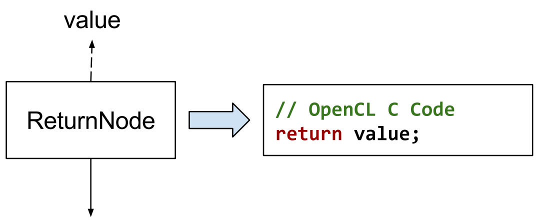


```
// Java code
for (int i = 0; i < 10; i++) {
    sum += i;
}
```

```
//Inside the lambda expression
int phi1 = 0;
int phi2 = 0;
for (; ; ;) {
   if (phi2 >= 10) {
       break;
   } else {
       int r1 = phi1 + phi2;
       int r2 = phi2 + 1;
       phi1 = r1;
       phi2 = r1;
    }
}
```

Figure 5.14: `LoopBeginNode` in the Graal IR and the generated sequential code for inner loops in OpenCL.

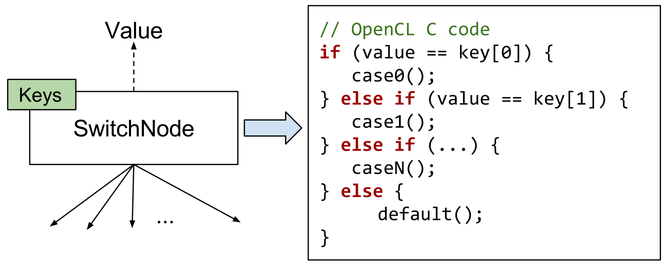
Return Node Return node is also part of the control flow in the Graal IR. The OpenCL code generation for this node is very simple. It just emits the **`return`** keyword and, if the optional value exists, its value. Figure 5.16 shows the Graal IR node on the left side and the generated OpenCL C code on the right.



```
// OpenCL C Code
return value;
```

Figure 5.15: `ReturnNode` in the Graal IR and the generated OpenCL C code.

Switch Node Marawacc supports the Java switch construction. It is mapped to a set of if-else statements in the OpenCL C code as shown in Figure 5.16 where the value is checked with each of its keys. The node has an input dependency that represents the value of the switch statement. Then it has an internal array to store all the possible values (keys). This node has multiple successors, one per case.

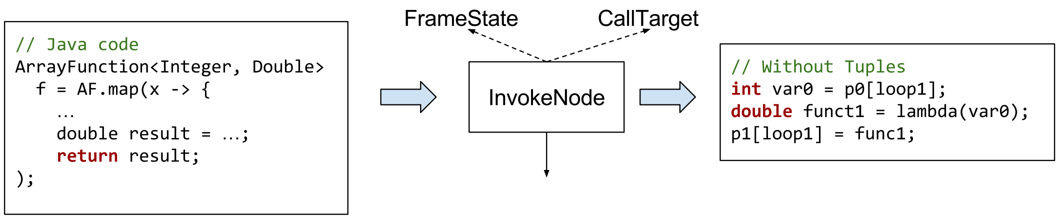


```
// OpenCL C code
if (value == key[0]) {
   case0();
} else if (value == key[1]) {
   case1();
} else if (...) {
   caseN();
} else {
     default();
}
```

Figure 5.16: `SwitchNode` in the Graal IR and the generated OpenCL C code.

Method Invoke Node Invoke nodes represent calls to methods. Marawacc differentiates two types of calls: the one that calls the lambda expressions and the rest of the calls. If the method to be invoked is not the lambda coming from JPAI, Marawacc obtains inputs and outputs types and generates the invocation call.

However, if the method to be invoked corresponds to the lambda expression, Marawacc specialises this invocation depending on the input and output types. If the input types to the lambda expression are primitive types (e.g. **`double`**) the corresponding variables are directly passed as arguments. Otherwise, if the data type of inputs or outputs are `Tuple`, then the code generator builds the C-struct before the invocation call.

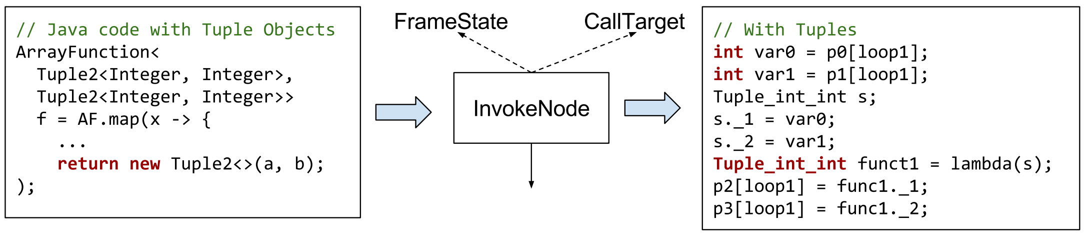


```
// Java code
ArrayFunction<Integer, Double>
  f = AF.map(x -> {
    ...
    double result = ...;
    return result;
);
```

```
// Without Tuples
int var0 = p0[loop1];
double funct1 = lambda(var0);
p1[loop1] = func1;
```

Figure 5.17: `InvokeNode` in the Graal IR and the generated OpenCL C code.

Figure 5.17 shows an example in Java, with its corresponding `InvokeNode` in the Graal IR and the OpenCL generated code. The node has two inputs, a reference to the method to be invoked (named *call-target*), and a frame state, that is used to deoptimise from the compiled to the interpreted code. The left side of the Figure 5.17 shows an example in Java where a map expression is created. The lambda expression computes a function from types `Integer` to `Double`. The right side of the Figure shows the corresponding OpenCL C code. Because of inputs to the lambda are passed as arrays, the code generator first obtains the input element, calls the lambda with that value and finally returns a new value that is used for the output array.



```
// Java code with Tuple Objects
ArrayFunction<
  Tuple2<Integer, Integer>,
  Tuple2<Integer, Integer>>
  f = AF.map(x -> {
    ...
    return new Tuple2<>(a, b);
);
```

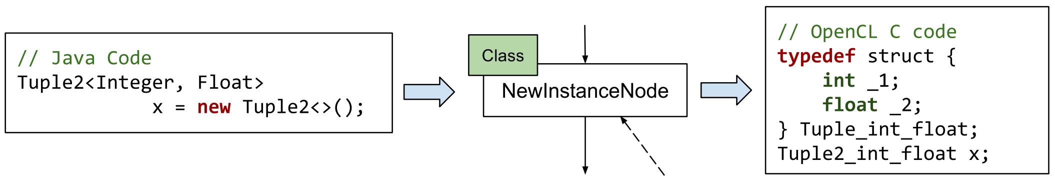
```
// With Tuples
int var0 = p0[loop1];
int var1 = p1[loop1];
Tuple_int_int s;
s._1 = var0;
s._2 = var1;
Tuple_int_int funct1 = lambda(s);
p2[loop1] = func1._1;
p3[loop1] = func1._2;
```

Figure 5.18: `InvokeNode` in the Graal IR and the generated OpenCL C code when using Tuples.

If the input to the lambda expression is a `Tuple` object, Marawacc has to build the tuple, which is a C-struct in OpenCL, and then it sets the fields and passes it as a parameter to the function. The Marawacc runtime, internally, stores each field in the Tuple in different arrays (as Section 5.5 will explain). This strategy allows the runtime to save the time for marshalling and unmarshalling the data. However, when a lambda expression is invoked with Tuples, Marawacc needs to reconstruct the struct that represents the tuple and passes it as argument to the lambda. Figure 5.18 shows this situation in which the lambda expression receives a `Tuple` and returns a new `Tuple`. In this example, the user function computes from types `Tuple2` to `Tuple2` of integers. The corresponding OpenCL code is shown on the right side of the Figure 5.18. First, the two values to build to tuple are obtained. Then, the tuple is declared and the values are set to the C-struct. The argument passed to the lambda is, in this case, a C-struct of type `Tuple2_int_int`. After the function call, the two values from the C-struct are set to the corresponding OpenCL output arrays.

Tuple Allocation Graal contains the node `NewInstanceNode` to allocate an object on the Java heap. Marawacc supports primitive data types and their wrappers (e.g. `Integer` , `Double` Java classes) and Java `Tuple` objects that are built for the JPAI array interface. Any other type of object is not currently supported in Marawacc.

Figure 5.19 shows a sketch of Java code that creates a new `Tuple2` object. The middle graph of the figure corresponds to the Graal IR node for a new object instance and the right shows the corresponding OpenCL C code. This IR node contains the information about the class of the object to be allocated. The dependency (dashed line) indicates that other IR nodes refer to the `NewInstanceNode` as a dependency. Marawacc takes this information and maps to the corresponding OpenCL struct.

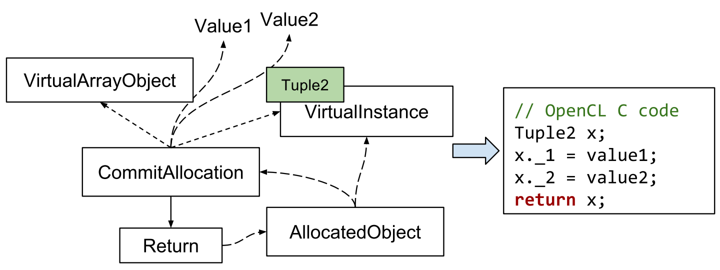


```
// Java Code
Tuple2<Integer, Float>
                  x = new Tuple2<>();
```

```
// OpenCL C code
typedef struct {
    int _1;
    float _2;
} Tuple_int_float;
Tuple2_int_float x;
```

Figure 5.19: `NewInstanceNode` in the Graal IR and the generated OpenCL C code.

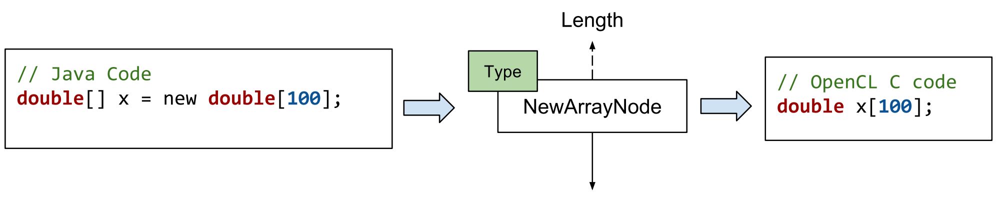
The Marawacc code generator also creates a `Tuple` C-struct for `CommitAllocationNode` . This node appears after the partial escape analysis optimisation [Stad 14b]. When Marawacc visits this node, it inspects its dependencies and emits the corresponding C-struct. Figure 5.20 shows an example that uses the `CommitAllocationNode`. This node receives a list of virtual objects and instance that specifies the *tuple* to be allocated. The right side of the Figure shows the corresponding OpenCL C code.



```
// OpenCL C code
Tuple2 x;
x._1 = value1;
x._2 = value2;
return x;
```

Figure 5.20: Example of handling IR nodes from partial escape analysis into OpenCL.

Static Array Allocation Marawacc also supports the static allocation of Java arrays in OpenCL. Graal IR contains the node `NewArrayNode` to indicate a new Java array allocation on the Java heap (Java arrays are also objects).



```
// Java Code
double[] x = new double[100];
```

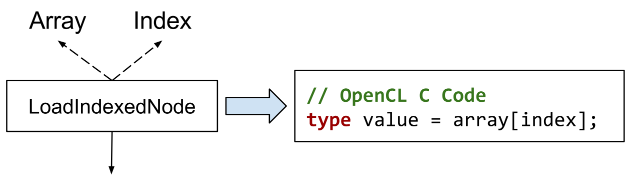
```
// OpenCL C code
double x[100];
```

Figure 5.21: `newArrayNode` in the Graal IR and the generated OpenCL C code.

Figure 5.21 shows a sketch of Java code on the left side with a new allocation of a double array. The corresponding IR node is represented in the middle of the figure.

The node has the length of the array as a data dependency. The right side of the figure shows the corresponding OpenCL C code for this static array allocation. Other related projects, such as Aparapi, can not handle static array allocation on GPUs.

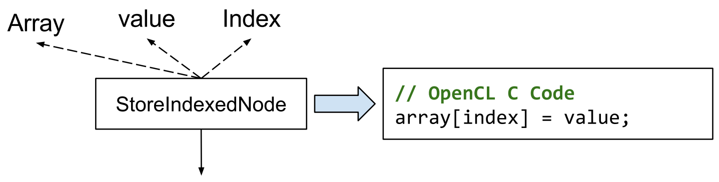
Load/Store Indexed Arrays Nodes Load and stores nodes values from/to arrays are fixed nodes in the Graal IR. Figures 5.22 and 5.23 show the corresponding Graal IR nodes for loading and storing values from/into an array in OpenCL global memory.



```
// OpenCL C Code
type value = array[index];
```

Figure 5.22: `LoadIndexNode` in the Graal IR and the generated OpenCL C code.

Before traversing the IR using the visitor pattern, Marawacc inspects all the arrays and inserts them into the the symbol table. When the load and store nodes are visited, the code generator looks up the variable name and emits the loading and storing for the corresponding nodes, as shown in Figures 5.22 and 5.23.

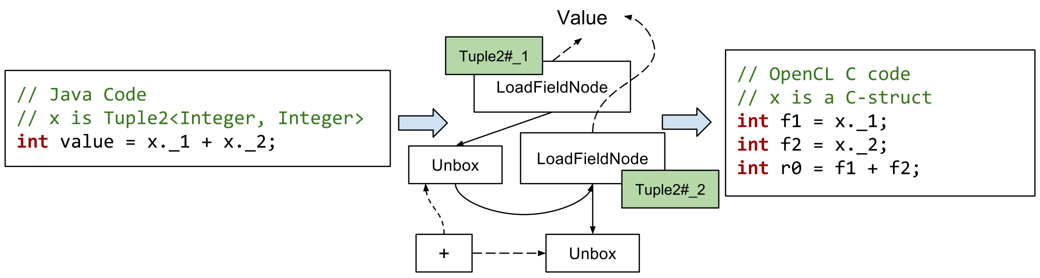


```
// OpenCL C Code
array[index] = value;
```

Figure 5.23: `StoreIndexNode` in the Graal IR and the generated OpenCL C code.

Load/Store Fields Nodes Loading and storing values into object fields is mapped to loading and storing those values into C-structs. The OpenCL code generator creates all the C-structs that are needed for code generation and emits them at the beginning of the generated code. The code also has an object symbol table for storing the corresponding association between Java and OpenCL (e.g. Java `Tuple2` class is mapped to a `struct2` C-struct with its corresponding types).

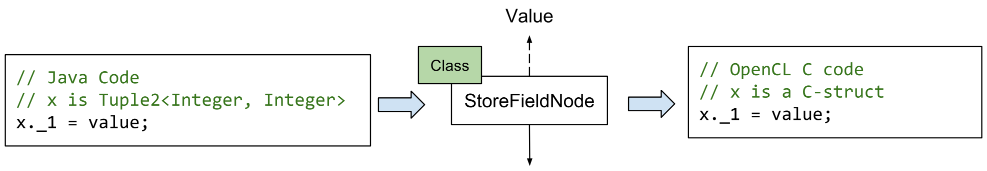
Figure 5.24 shows an example of use of `LoadFieldNode`. The left side corresponds to an example in Java that adds two values from fields of a `Tuple2` object. The graph in the middle shows the Graal IR for the Java code. It contains two `loadFieldNodes`; one for each field. The right side of the figure shows the corresponding OpenCL C code. It



```
// Java Code
// x is Tuple2<Integer, Integer>
int value = x._1 + x._2;
```

```
// OpenCL C code
// x is a C-struct
int f1 = x._1;
int f2 = x._2;
int r0 = f1 + f2;
```

Figure 5.24: `LoadFieldNode` in the Graal IR and the generated OpenCL C code.



```
// Java Code
// x is Tuple2<Integer, Integer>
x._1 = value;
```

```
// OpenCL C code
// x is a C-struct
x._1 = value;
```

Figure 5.25: `StoreFieldNode` in the Graal IR and the generated OpenCL C code.

first loads the two variables into intermediate ones (`f1` and `f2`) and then it performs the addition.

Storing a field in OpenCL is very similar to the loading just explained. Marawacc uses the object table information stored at the beginning of the code generation process. Figure 5.25 shows an example of a Java program that stores a value into the field `_1` of the `Tuple2` Java class. The graph in the middle illustrates the corresponding IR node and the snippet code on the right shows the corresponding OpenCL C code.

5.5 OpenCL-Java Runtime for Data Management

This section discusses and addresses the challenges and the importance of managing data efficiently in a heterogeneous setting with Java and the CPU on the one side and OpenCL and the GPU on the other. Firstly, this section describes the problem with data transformations and its impact on performance when programming heterogeneous architectures from managed programming languages. Secondly, it presents an optimisation to efficiently manage data transformation between Java and OpenCL.

5.5.1 The Challenge of Data Management

Efficient data management is essential when programming heterogeneous architectures from managed programming languages like Java. Usually CPUs and GPUs do not share the same physical address space and, therefore, data has to be moved between them which can easily limit the performance benefits of the GPU. The usage of Java introduces an additional challenge for efficiently managing data. Java uses *reference* *semantics*, i.e., references to objects are managed rather than the objects itself.

When a collection of objects is stored in an array, the references to the objects are stored in the array. This data management is fundamentally different from *value semantics* used in OpenCL (which inherits from C) where not references to objects but instead the objects themselves are stored in the array.

For primitive types like `int` and `float`, Java uses the same value semantics as C, but, unfortunately, these types cannot be used together with Java generics. JPAI is implemented with Java generics for ensuring strong type safety. This prevents the use of primitive types such as an array of `int`. To circumvent this drawback, Java provides corresponding wrapper types (`Integer` and `Float`) which follow Java's conventional reference semantics and can be used together with generics. Therefore, before running the OpenCL kernel on the GPU, Marawacc has to first transform the Integer array object into an array of primitives of `int`s, then to transfer the data from the host to the device and finally to execute the OpenCL C program on the device.

The transformation process between the data types in Java and OpenCL is known as *marshalling*. Unfortunately, marshalling is a very time consuming task. To illustrate how expensive this process is, a detailed profiling trace of one benchmark that uses `Tuples` running on an Intel i7 CPU and a AMD GPU is reported.

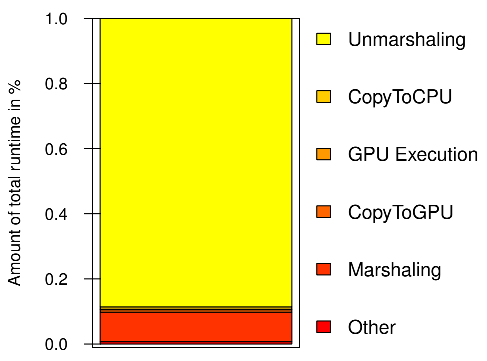


Figure 5.26: Breakdown of the total runtime of the Black-Scholes executed on an AMD GPU. The marshalling and unmarshalling steps take over 90% of the total runtime.

Figure 5.26 shows the runtime for the *Black-Scholes* application implemented using JPAI as described in 4.6. The total runtime is divided into six subsections that correspond to each step needed to execute a Java application on the GPU. The breakdown of the runtime corresponds to the time that it takes to marshal the data, transfer the data from host to device (`CopyToGPU`), execute on the GPU, transfer the data from the device to the host (`CopyToCPU`) and *unmarshall* the data, which transforms the OpenCL output representation into a Java array. `Other` represents the time spending in the rest of the VM logic, such as method dispatch and bytecode interpretation that it is not included in the rest of the timers. As it can be seen, the marshalling and the unmarshalling together take up 90% of the total runtime.

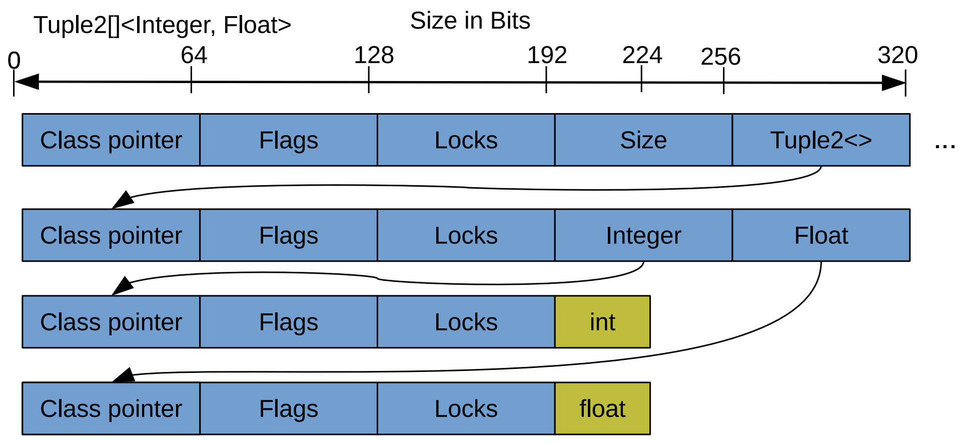


Figure 5.28: Anatomy of Java Objects

Marshalling in Java Why is the cost of data transformation so high? To answer this question, deeper analysis about how the JVM represents objects in memory is needed.

Figure 5.27 shows the Java anatomy for an array of Tuple2 objects. Each object in Java has a header which contains information, such as the class pointer and synchronization flags. The left part of the figure shows the array representation. Each element in the array corresponds to a Tuple2 array. The middle object represents a Tuple2 of Integer and Float objects. Each element in this object contains other two objects, one for Integer and one for Float. The right part of Figure 5.27 shows these two objects' representation. These two wrapper classes contain the corresponding **int** and **float** respectively.

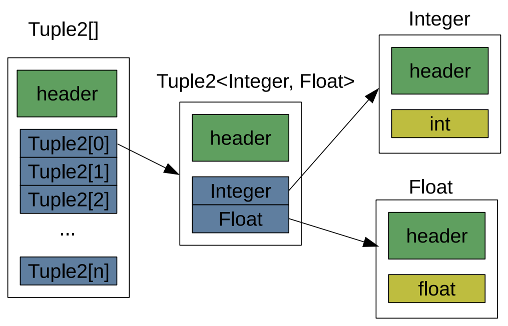


Figure 5.27: Anatomy of Java Objects

Figure 5.28 shows the Java object representation in memory for the JVM. The top of the figure represents a Java object of type `Tuple<Integer, Float>` in memory. The object is in a 64-bit system. The first 64 bits represents a pointer to the class object. In this case it is a pointer to `Tuple2.`**`class`**. The `flags` field indicates the state of the object and whether the current object is an array or not. The `lock` field indicates the object synchronization. From that point, bit 192, the object has the fields, that are basically two other objects (Integer and Float). Each object has the same layout except for the last part where the real data is stored in 32 bit field to store either the **int** or the **float**.

Marawacc targets GPUs because it can speed-up compute intensive and data parallel applications. However, just because the data layout has to be transformed between

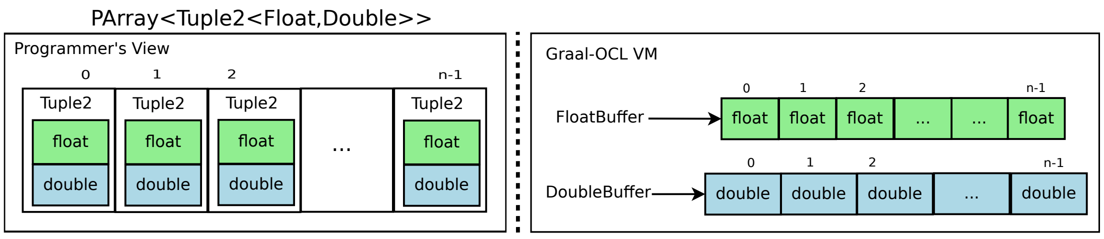


Figure 5.29: Strategy to avoid marshalling in the VM for the OpenCL component.

different programming languages representations, the total runtime can be degraded and GPUs could not be as attractive for many developers as it should be. This is clearly a huge issue when using heterogeneous programming from managed languages like Java. The rest of this section describes a runtime optimisation to address this issue and execute a Java application as competitively as native OpenCL implementations.

5.5.2 Avoiding Marshalling: A Portable Array Class

For data parallel and iterative applications for large input data sizes, the data management becomes the bottleneck when using heterogeneous hardware such as GPUs. Developers want to use GPUs because they offer good performance at a reasonable price, in terms of original investment, maintenance and energy consumption.

To address this issue, Marawacc implements a new data structure to avoid the marshalling. This data structure is an array named **PArray** for Portable Array (portable across devices), that essentially creates a view of the original array in the OpenCL representation from the Java object. Figure 5.29 shows a representation of the `PArray`. The left side of the figure shows the programmer's point of view and the right side shows the VM point of view, which is executed using the Graal VM. This figure shows an example with `PArray` of `Tuple2<Float, Double>` in the left side. What developers create is an array of Tuple2, but, when an `add` or `append` operation is performed, it stores the unboxed values directly into the `ByteBuffers`. When the marshal and unmarshall are required, Marawacc will only pass the reference to the `ByteBuffers`. The `PArray` class follows a value semantics, i.e., instead of references to values copies of the values are stored in the array.

PArray uses Java Generics The `PArray<T>` class uses the *Java Reflection API* to inspect the type of `T` the first time the array is filled up with data. If `T` is a class such as `Float`, the Marawacc runtime directly stores a buffer of the underlying primitive type; for an `PArray<Float>` internally Marawacc stores a Java `FloatBuffer` from the `Java.nio`

```
1 PArray<Integer> input = new PArray<>(size);
2 for (int i = 0; i < size; ++i) {
3         input.put(i, i);
4 }
```

Listing 5.3: Example of PArray

package which is a low-level data storage of primitive `float` values. This mechanism helps Marawacc to circumvent the drawback that Java generics cannot be used together with primitive types.

The low-level buffers, like `FloatBuffer`, store their values in the same data layout that OpenCL C expects, but they are managed as the high-level language requires. When executing an OpenCL kernel, Marawacc directly transfers the data from the `FloatBuffer` to the GPU without the marshalling step, saving an important part of the whole runtime. The class `PArray` provides the custom methods `get` and `put` for accessing the elements in the arrays.

When a lambda expression receives a `tuple` as a parameter, Marawacc reads the input values from multiple OpenCL buffers. There are as many buffers as the `tuple` degree (e.g. `Tuple3` reads from three different OpenCL buffers). If the lambda expression returns a `tuple`, Marawacc allocates the corresponding buffers for writing the output. The Marawacc code generator does not handle any type of Java object, but only Tuples, numeric wrappers for primitive data types and primitive arrays. Marawacc OpenCL code generator knows the layout of the `tuple` and maps the Java class into an OpenCL `struct` with functions for accessing the fields. When copied back from the GPU to the CPU, these OpenCL buffers are copied directly into the allocated low-level Java buffers, thus, avoiding the unmarshalling.

Listing 5.3 shows how to program with PArray data structure. Line 1 creates the PArray passing the size and the type. The type is necessary because the `PArray` uses generics and the types are erased at runtime. Lines 2-4 initialize the `PArray`. When a new element is added using the `put` method, the runtime sets the element into an internal `ByteBuffer` array, which is, in this case, a Java `IntBuffer`.

Tuples Object Support For arrays of tuples, Marawacc stores multiple byte-buffers internally, one for each component of the tuple, since this layout is generally more efficient on accelerators. For instance, if a `PArray<Tuple2<Float, Double»` is used, Marawacc stores one `FloatBuffer` and one `DoubleBuffer` as can be seen in Figure 5.29.

When an element of such an array is accessed from Java, Marawacc builds a `Tuple2<Float, Double>` object at runtime.

OpenCL Data Optimisations: Pinned Memory The use of the `PArray` not only reduces the cost of marshalling, but also creates opportunities for extra optimisations. When programming with heterogeneous systems, data transfers from the host to the device ($H \rightarrow D$) and vice-versa ($D \rightarrow H$). When using GPUs, this process can be very slow. GPUs have their own memory and memory address translation hardware therefore the OS has to manage all request accesses to the GPU. However, there is a way to speed-up the translation processes by using *pinned-memory* or *page-locked* memory by increasing the speed of the Direct Memory Address (DMA) accesses. CPUs implement virtual memory to allow the system to protect the memory when running multiple applications at the same time. It also allows programmers to use more memory than the real address space available. When the CPU issues a virtual address, the operating system has to translate that virtual address into a physical address. When performing a copy from the CPU to the GPU, this translation process between the CPU memory and the GPU has to be done too. To accelerate the translation process, a page can be locked, which means that it can not be moved to disk if needed. As an advantage, because the memory pages are pinned, it can not violate any address and it is limited to the assigned memory address space. Therefore, the DMA speeds-up the memory transfer between the CPU and the GPU.

When using the `PArray`, Marawacc can create GPU pinned memory transparently, and, therefore, the speed of the memory transfers between GPUs and CPUs is increased. In the case where the computation involves the use of an OpenCL device (e.g., GPU), Marawacc allocates the data in *pinned memory* at the OS level (i.e., the memory pages involved will never be swapped). This enables faster transfer time between the host and the device since the GPU DMA engine can be used, freeing the CPU from managing the data transfer.

Summary This section has described the challenges and issues of data transformations between Java and OpenCL (marshalling). To address those issues, a mechanism to avoid the cost of marshalling and unmarshalling has been proposed by using a new data structure, the `PArray`. By using the `PArray`, programmers do only eliminate the cost of marshalling, but also create opportunities for hardware specific optimisations such the use of the GPU *pinned memory*. The effect in the overall performance of this optimisation is presented in the next sections.

```
for i = 0 to 100 do
    call Garbage Collection
    start = get time;
    run code on GPU;
    end = get time;
    record time (end - start) ;
end
```

Algorithm 2: Logic for measuring in Java

5.6 Evaluation

This section presents the experimental evaluation of the Marawacc OpenCL JIT compiler framework and the memory optimiser. This section first describes the experimental setup. Then it shows performance results using two different GPUs. Finally, a detailed comparison between Marawacc, Aparapi, JOCL and native OpenCL is presented.

5.6.1 Experimental Setup

Marawacc is evaluated using two different GPUs: an AMD Radeon R9 295X2 GPU with $2 \times$ 4GB of graphics memory and a Nvidia GeForce GTX Titan Black with 6GB graphics memory. The drivers used were the latest available (AMD: 1598.5, Nvidia: 331.79). The Java version used is Oracle JRE 8 (1.8.0.65).

The times reported for all the benchmarks are measured in peak performance, which means that the time reported is when JVM compiles the code after the initial warming up. When running a program in a VM, many threads can run at the same time, like the garbage collector, threads for compilation in background, etc. We are interested in the time that takes to solve the overall applications on the GPU, including the runtime that manages the GPU application. In order to reduce the noise in the measurements, the methodology described in Algorithm 2 has been taken to report performance and to compare it with another parallel frameworks such as C++, Aparapi and JOCL. It executes an input program 100 times. Before calling the GPU code, it invokes the garbage collector (GC) to be sure that there is enough space on the Java heap to not interrupt the process due to a GC collection. The benchmarks were executed with the JVM option `-XX:+PrintGCDetails` enabled, which allows to track if the GC works in the middle of the compute kernel execution. If during the program execution the GC interrupts the computation, that execution were discarded for computing the median values. However, for the experiments reported, the GC did not interrupt any of the executions, apart from the explicit call to the Java's garbage collector before start measuring, as shown in Algorithm 2. Therefore, the 100 execution were encountered to compute the median.

To evaluate Marawacc, five different applications from different domains were executed. The applications used are detailed in Section 4.6. Table 4.1 shows the data sizes used for the benchmarks.

Code Compilation Generating OpenCL code with Marawacc takes from 70 to 300 milliseconds. Once Marawacc generates and compiles the generated OpenCL C kernel, it stores the binary into a cache that the Marawacc runtime handles. In contrast with similar approaches such as Sumatra and Stream API, Marawacc reuses the same compiled kernel if the input lambda expression is invoked again.

Variability The median value is reported to represent peak performance. There is a variation between 0.1% and 4.9% from the median value for the rest of 99 executions. This variation is small enough to consider the median value a good value to represent peak performance.

5.6.2 Generated GPU Code Quality

This section shows a performance evaluation of the Marawacc OpenCL C kernel that is generated from the Java bytecode using the Graal compiler. Marawacc is compared to JOCL and the native code (OpenCL C++).

Figure 5.30 shows the runtime, in nanoseconds, for the OpenCL kernel generated by the Marawacc OpenCL JIT compiler, JOCL and OpenCL C++. The x-axis shows the corresponding benchmark and the y-axis shows its runtime for the largest data size shown in Table 4.1. In this figure, the lower, the better. In general, Marawacc is 12% slower than JOCL and the native code on AMD GPU and 56% slower on NVIDIA GPU. For data intensive applications, such as NBody and Montecarlo, the Marawacc OpenCL code is one order of magnitude slower than the native code on the NVIDIA GPU. However, for other benchmarks such as Saxpy and KMeans the OpenCL kernel is within 98% of the native code. Note that OpenCL kernels are executed in a few milliseconds, such as Saxpy, KMeans and Blackscholes, where the OpenCL kernel execution is just a tiny fraction of the total time (as Section 5.6.5 will show). For more

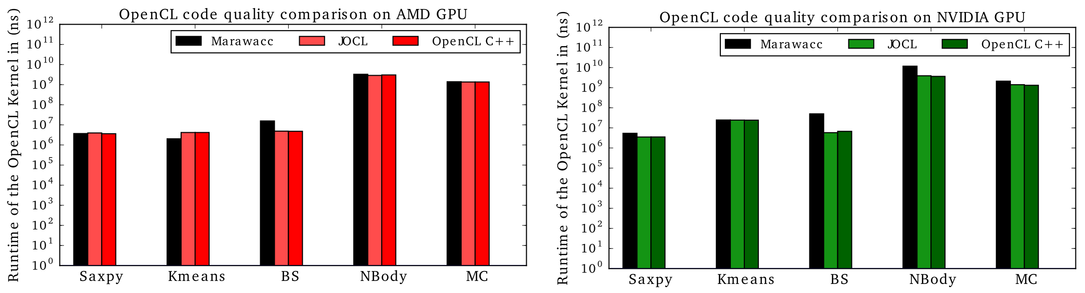


Figure 5.30: OpenCL Kernel quality of the Marawacc OpenCL code generator. X-axis shows the benchmarks and y-axis shows total runtime in nanoseconds represented in logarithmic scale.

details, Table A1 and Table A2 show the total runtime in nanoseconds for all the benchmarks. These tables show that, for Saxpy, Kmeans and Blackscholes, the Marawacc total runtime is between 40 and 80 milliseconds. For these benchmarks, optimising data transfers and data transformation is much more important than optimising the kernel, as Section 5.6.3 will show.

There is one benchmark that the generated OpenCL C code with Marawacc is 2x faster than the native code. This is the KMeans only for the AMD GPU. This benchmark has been investigated and it has shown that, before generating the OpenCL C code from the input lambda expression, the Graal compiler obtains one of the parameters of the loop in Kmeans as a constant. Since the Marawacc OpenCL JIT compiler optimises the CFG by applying architecture independent optimisations, such as constant propagation, the loop is easily unrolled by the driver OpenCL compiler. This is one of the benefits of compiling a program at runtime.

In general, Marawacc generates naive kernels that can be optimised by using, for example, local and constant memory. However, for many potential applications that can be executed on GPUs, the time that the compute kernel takes is not significant enough compared to the data transformations and runtime management from Java. This thesis also optimises the runtime and data transformation as well as reduce the cost of marshalling to increase the overall performance of Java applications when running on GPUs.

5.6.3 Runtime Results for the GPU Execution

Figure 5.31 shows the speedups of Marawacc over sequential Java when performing the computations on two different GPUs. Figure 5.31 uses a logarithmic scale, because of the large speedups obtained for some of the benchmarks. For each combination of

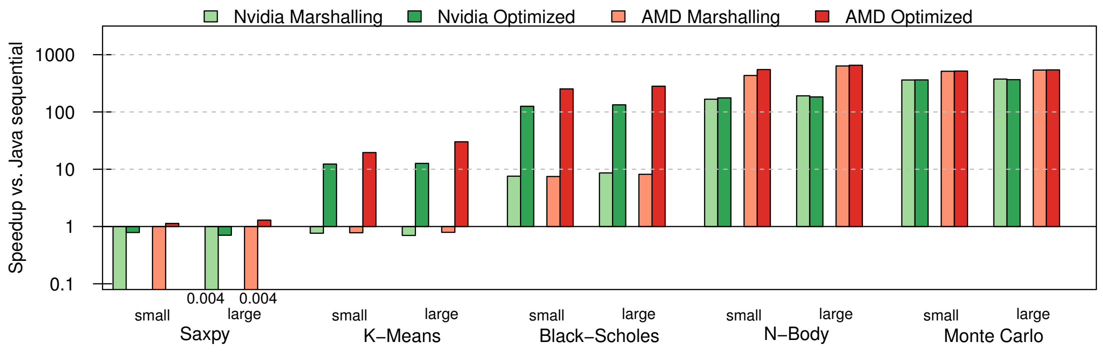


Figure 5.31: Speedup for all benchmarks over sequential Java execution with GPU.

benchmark, the input size and GPU, two bars are showed: the left one showing the runtime with marshalling and the right bar showing the runtime `PArray` data optimisations enabled to avoid marshalling.

The effect of marshalling is very significant for some benchmarks and, by using the optimised `PArray` to avoid marshalling, it enables large additional speedups. For the *saxpy* application the GPU execution with marshalling introduces a significant slowdown over sequential Java execution. By avoiding marshalling, Marawacc can even obtain a small speedup on the AMD GPU for this memory-intensive benchmark.

Avoiding marshalling for *K*-Means allows the speedups to increase by a factor of 33×, from about 0.9× to 30×. For Black-Scholes the speedup improves from 8.1× to 281× for the large data size on AMD, a 35× improvement. These two applications are examples of applications which offer substantial parallelism, but require a significant amount of data to be transferred between Java and the GPU. For these types of applications, the optimised `PArray` offers significant runtime improvements.

The last two applications N-Body and Monte Carlo are very computational intensive, therefore, these applications harness very effectively the power of the GPU resulting in massive speedups of up to 645 over sequential Java. For these applications, little data is moved between Java and the GPU, therefore, the Marawacc runtime data optimisations has only a minor effect on the overall application runtime. Additionally, the use of the optimised `PArray` to avoid marshalling never decreases overall performance of the applications.

Overall, these results show that Marawacc offers high performance for data parallel application executed on GPUs. In addition, for an important class of applications, the proposed data optimisations significantly improve performance.

5.6.4 Comparison with State-of-the-art Java GPU Computing Solutions and OpenCL

This section investigates the performance of Marawacc against two state-of-the-art frameworks for Java GPU programming, *JOCL* and *Aparapi*, as well as native OpenCL C++ implementations of the benchmarks presented in Section 4.6. This section first gives an overview of JOCL and Aparapi GPU frameworks (introduced in Chapter 3) and then it shows the performance comparison.

JOCL The *Java binding for OpenCL* (JOCL) is a Java library wrapping all calls to the OpenCL API. This allows programmers to write OpenCL applications from Java. GPU computations are written as native OpenCL C code embedded as strings in Java. Then, the wrapper takes the C code represented as a String and compiles to the GPU binary format using the OpenCL driver available.

The provided API in JOCL uses the function names of the OpenCL standard. This is good for the programmer that previously have learnt OpenCL, because the heterogeneous Java application will be very similar to the equivalent OpenCL C or C++ program. The complexity of implementing Java applications with JOCL is very similar to implementing a OpenCL applications with C or C++. Programmers need to create the OpenCL platform, devices, context, command queue, buffers, compile the kernel, etc.

AMD Aparapi AMD Aparapi [AMD 16], or simply Aparapi, is a framework to develop heterogeneous applications within Java. Similar to Marawacc, it compiles a subset of Java to OpenCL at runtime. However, the Aparapi programming model remains low-level as it resembles the abstractions of OpenCL requiring experts to exploit parallelism explicitly.

Aparapi provides an API and a set of classes that programmers must use in order to compile and to execute a Java application into a GPU via OpenCL. There are several restrictions in Aparapi. For instance, there is no support for objects, apart from 1D Java arrays. There is no support for exceptions, or inheritance, even though the computation is provided in a class.

Performance Comparison Figures 5.32 and 5.33 show speedup over sequential Java on the AMD and Nvidia GPUs respectively. The leftmost bar shows the speedup of Marawacc. The second bar shows Aparapi. The third bar shows JOCL and the last one shows the speedup with native OpenCL C++.

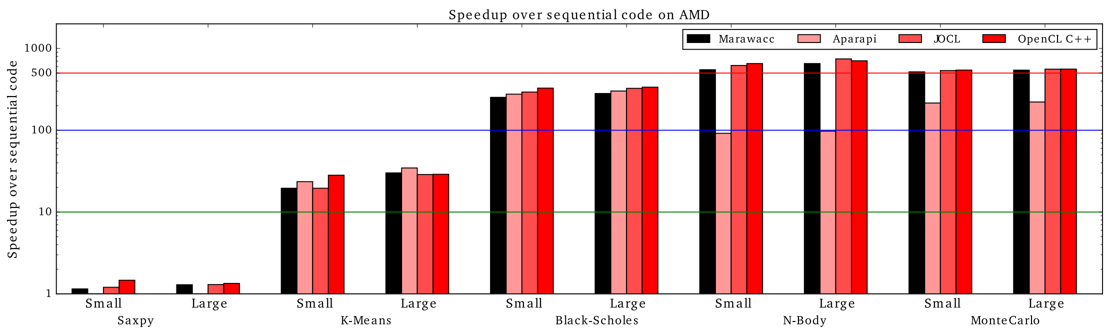


Figure 5.32: Speedup over sequential Java of the Marawacc framework compared against three other GPU implementations running on an AMD Radeon R9.

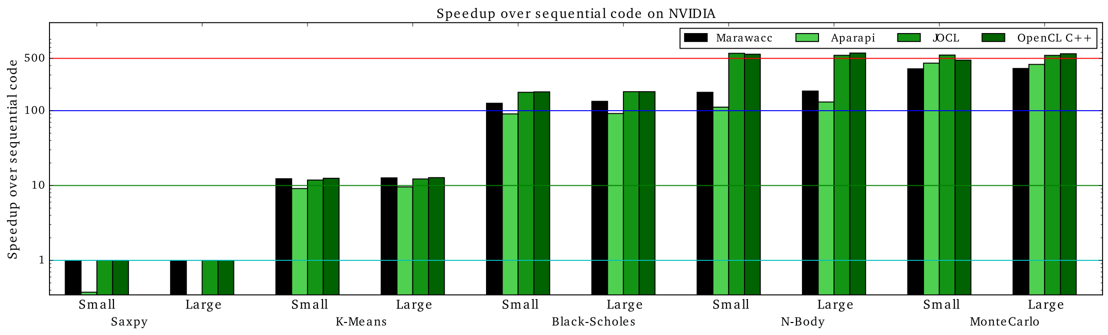


Figure 5.33: Speedup over sequential Java of the Marawacc framework compared against three other GPU implementations running on an NVIDIA Titan Black.

The results clearly show that Marawacc achieves performance close to JOCL and native OpenCL on both platforms and for most benchmarks. The performance of Marawacc is better than the performance of Aparapi for almost all the cases.

On average Marawacc is only 23% slower than native OpenCL code. For some benchmarks, such as `Nbody`, the native implementation applies optimisations to the kernel which are not yet supported by Marawacc, such as the usage of local memory. On the Nvidia GPU this leads to a speedup of 589× for the OpenCL implementation versus a speedup of 183× for Marawacc. For more detailed information, Table A1 and Table A2 in Appendix A show the total runtime for all the benchmarks and implementations.

Comparing Marawacc to the high-level Aparapi framework, Marawacc achieves better performance on almost all benchmarks on both GPUs. This is partly due to the use of *pinned memory*, which is not performed by Aparapi. The results also suggest that Marawacc generates more efficient kernel code, as it has a performance advantage of up to 1.74x (74%) over Aparapi on the AMD platform for the compute intensive benchmarks `N-Body` and `MonteCarlo`. On the NVIDIA GPU, Marawacc performs up

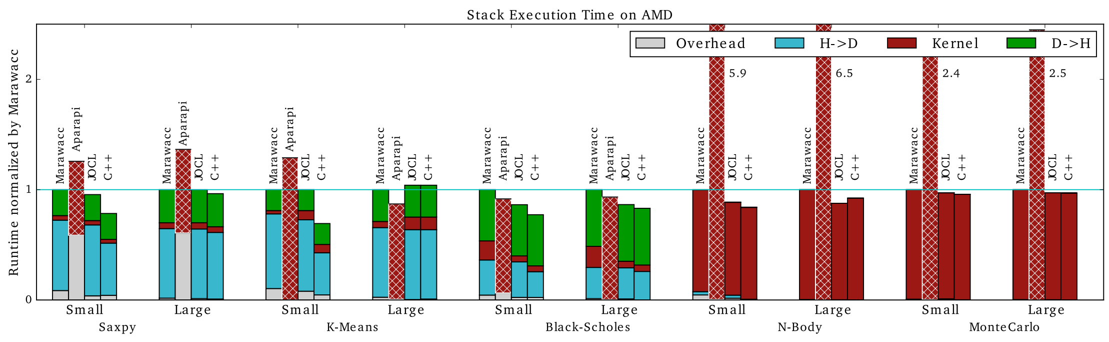


Figure 5.34: Breakdown comparison of the execution time for the AMD ATI GPU with Marawacc framework. It is normalized to the Marawacc's execution time. The lower the better.

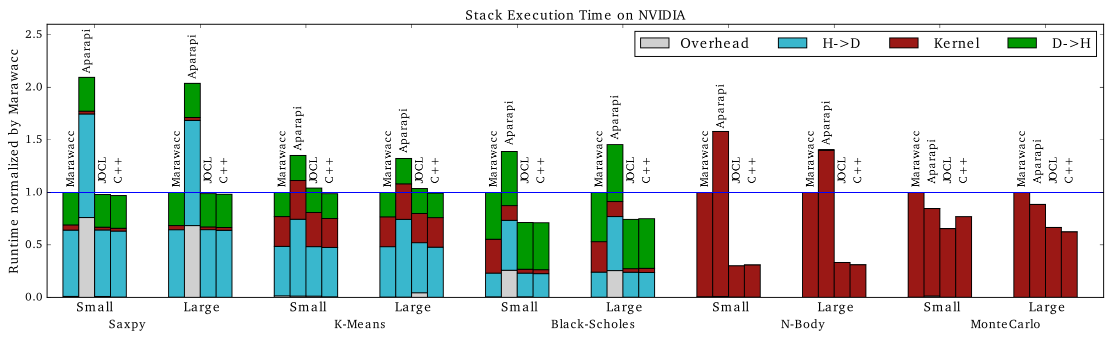


Figure 5.35: Breakdown comparison of the execution time for the NVIDIA GPU with Marawacc framework. It is normalized to the Marawacc's execution time. The lower the better.

to 1.38x over Aparapi. On average, Marawacc performs 55% better than Aparapi for the selected benchmarks.

5.6.5 GPU Execution Comparison with State-of-the-art GPU Computing Solutions in Java

Figures 5.34 and 5.35 show a breakdown of the execution times of Marawacc, Aparapi, JOCL and OpenCL C++ on AMD and Nvidia GPUs respectively. The execution time is divided in four sections: copy data from host to device ($H \rightarrow D$), kernel execution, copy data to host ($D \rightarrow H$) and the *rest* of the runtime correspond to the time not spent on the device. The first three timers are measured using the OpenCL profiling API. The y-axis shows runtime normalized to Marawacc, therefore, runtimes below 1.0 are faster than Marawacc.

Aparapi uses different OpenCL functions for copying data to and from the GPU. Unfortunately, with the AMD OpenCL implementation used for this evaluation, the corresponding OpenCL timers did not distinguish between the data transfer time and the kernel execution time. Therefore, the OpenCL timers are combined into a single stripped bar for Aparapi on AMD in Figure 5.34.

The first three benchmarks spend a large portion of the runtime with data transfer and only a small fraction of the runtime is dedicated to the actual computation. In Figure 5.35 we see that one of the main implementation differences between Marawacc and Aparapi resides in the data transfer. Aparapi spends a significant larger amount of the runtime transferring data compared to Marawacc. In contrast to Aparai, Marawacc uses pinned memory by default and also avoids marshalling by using the `PArrays` data structure. Aparapi also performs an unnecessary copy of data to the GPU filling the output array with default data which is never read from. Marawacc avoids this copy.

The data transfers for the `saxpy` and `blackscholes` benchmarks are executed is as fast as JOCL and OpenCL C++. That means that Marawacc does not introduce any overhead and is as fast as native code.

As was shown in the previous section, the last two applications are compute-intensive. And almost all runtime is spent in the kernel execution. The optimisations of data transfers are, therefore, not as significant for these benchmarks. Speedups that Marawacc obtained range from $0.83\times$ to $6.70\times$ faster compared to Aparapi for these benchmarks on the AMD GPU and $0.84\times$ to $2.10\times$ faster on the NVIDIA GPU.

5.6.6 Productivity Comparison

To provide a productivity comparison, Table 5.1 shows an analysis of the number of lines of code required in each parallel programming approach to implement each application as an indicator for the productivity of programmers. Only the lines that are essential for writing the application are counted, while not counting empty lines but including code to deal with data allocation on the host side. In OpenCL C++ and JOCL, the user needs to write the C++/Java program including management calls for the GPU communication. Moreover, programmers need to write the OpenCL kernel explicitly in a separated file or in a string.

In the case of Aparapi and Marawacc, applications are entirely written in Java. In Aparapi, the computation is expressed by extending the `Kernel` class and overriding the `run` method. In Marawacc, the kernel code is written as a Java high-level lambda expression.

	OpenCL C++			JOCL			Aparapi	Marawacc
Application	C++	OpenCL	Total	Java	OpenCL	Total	Java	Java
Saxpy	81	7	88	107	7	114	41	21
Black-Scholes	81	68	149	111	68	179	80	64[1]
Kmeans	94	31	115	119	31	150	52	40
NBody	104	40	144	127	40	167	85	67
MonteCarlo	73	20	93	95	20	115	37	35

Table 5.1: Lines of code comparison per benchmark. The number given in the table are the lines of code with the function calls not inlined manually.
[1]Due to a Graal inliner bug we had to manually inline function calls resulting in 89 lines of code.

As Table 5.1 shows, Marawacc requires significant fewer lines of code than OpenCL C++ and JOCL, but offers similar performance as shown in Section 5.6.4. It avoids the boilerplate code required by the low-level programming models to manage the GPU communication. When compared to Aparapi, Marawacc requires a similar number of lines of code with shorter programs due to the more concise lambda expressions used with the JPAI interface.

5.6.7 Feature Comparison for GPU Programming in Java

Table 5.2 summarizes important features for all compared Java frameworks for GPU programming with OpenCL and the GPU frameworks for Java presented in Chapter 3. The columns indicate, if the OpenCL kernel is generated at runtime (JIT), the data layout is optimised to avoid marshaling, data transfers to and from the GPU are optimised, if arrays can be allocated in the kernel code, if Java Objects are supported, if deoptimisations to Java exists if the GPU execution fails, and if Java lambdas expressions are supported. Out of the investigated frameworks, Marawacc is the only one that supports all the features with the exceptions of any type of object (it only supports the wrappers for primitive types, such as `Integer`, and the Java `Tuple` class).

5.7 Summary

This chapter has presented Marawacc, a GPU JIT compiler and a runtime system for GPU execution and data optimisations within Java.

Work	JIT	Data Layout Opt.	Transfer Opt.	Array Alloc.	Objects	Deopt.	Lambda
Marawacc	Yes	Yes	Yes	Yes	Partial[1]	Yes	Yes
JOCL	No	Manual	Manual	No	No	No	No
Aparapi	Yes	No	No	No	No	Yes	Yes
Sumatra	Yes	No	No	Yes	No	No	Yes
[Ishi 15]	Yes	No	Yes	No	Yes	Yes	Yes
RootBeer	No	No	No	Yes	Yes	No	No
JaBEE	Yes	No	Yes	Yes	Yes	No	No
JaCC	Yes	No	Yes	Yes	Yes	Yes	No

Table 5.2: Features comparison for GPU programming in Java.
[1]Tuples, arrays and numeric objects for the primitive data types.

The Marawacc GPU JIT compiler transforms, at runtime, Java bytecode to OpenCL C using the Graal compiler and compiles the final generated kernel using the OpenCL C driver. The Marawacc runtime manages the GPU execution and optimises the data transformation between Java and OpenCL. To optimise the marshalling process, the runtime exposes the `PArray` data structure to programmers, allowing it to improve the overall performance.

This chapter has also evaluated the Marawacc approach using five benchmarks from different application domains. It has shown that, by using Marawacc with GPU, speedups of over 645$\times$ compared to sequential Java are possible. The evaluation section has also shown that Marawacc data optimisations avoid the cost of marshalling and they can substantially improve the overall performance by up to 35x when executing on GPUs.

It has also demonstrated that Marawacc framework offers performance competitive to Aparapi, JOCL and OpenCL C++. Marawacc is, on average, 23% slower than OpenCL C++, ranging from 30% of the performance to absolutely no overhead. By using the Marawacc compiler and runtime framework, speedups of up to 1.74x can be achieved over Aparapi. On average, Marawacc performs 55% better than Aparapi on both GPUs. Moreover, Marawacc is more concise, allowing programmers to be more productive and to write the same program with less lines of code.

The next chapter shows how Marawacc can be exposed to other JVM languages in order to execute programs implemented in dynamically typed languages on GPUs.

Chapter 6

GPU JIT Compilation of Dynamically Typed Languages

The Java Virtual Machine (JVM) is an abstract machine that executes Java bytecode. The JVM was originally designed for the Java programming language, allowing to execute applications that follow the Java semantics and Java memory model. Java bytecode, as explained in Section 2.4.1, is an intermediate representation that can also be used for other programming languages (called guest programming languages for the JVM).

Once a compiler translates the guest source program into the Java bytecode, the JVM treats all applications as equally independent of the programming language with the benefits of using the same advance JIT compiler and all its optimisations. JRuby, Scala, Clousure and JPython are examples of guest programming languages that are built on the JVM. A similar example is the Microsoft Common Language Runtime (CLR), an application virtual machine that executes multiple applications under the same VM using the .NET Framework.

One of the main difficulties of developing a Virtual Machine (VM) and an optimising compiler from scratch is that it is a very costly process, which requires a lot of engineering effort. By executing new languages within an existing and optimised virtual machine infrastructure, many optimisations can be performed for the guest languages at no extra cost for language developers.

This thesis is focused on the JVM and in the prior chapters, it has focused on Java applications. As seen in Chapter 5, the proposed technology to execute Java applications on heterogeneous hardware is implemented in the JVM. This chapter leverages this approach to guest dynamically typed programming languages running on top of the JVM. This chapter first shows the challenges of implementing a GPU JIT compiler for dynamic and interpreted programming languages and then it presents a set of compiler techniques to automatically offload applications written in guest programming languages on top of the JVM to the GPU. This chapter refers to the term *dynamic programming languages* in the context of high-level dynamically typed and interpreted languages such as R and Ruby.

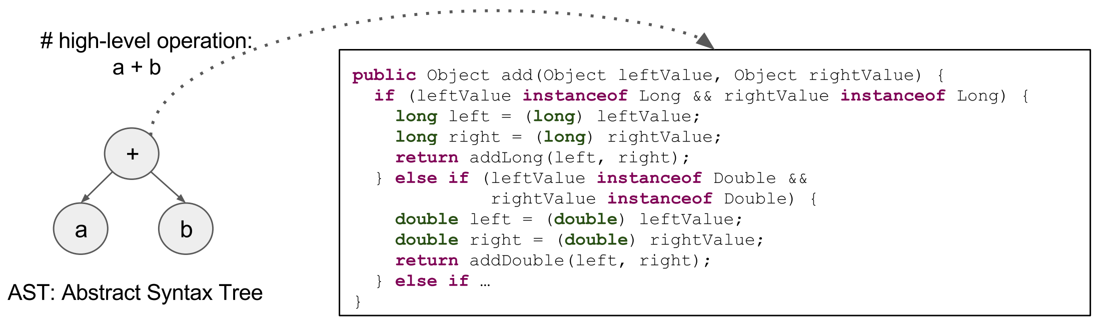
This chapter is organised as follows: Section 6.1 presents the challenges of generating OpenCL code from high-level and dynamic programming languages at runtime. Section 6.2 presents an overview of the proposed compiler techniques to generate OpenCL code using the R guest programming language as a use-case. Section 6.3 presents how the OpenCL code generation is performed. Section 6.4 presents optimisations that the compiler framework performs before generating the OpenCL C code that help to improve the overall performance. Section 6.5 presents a technique to deoptimise (fall back) from the optimised GPU compiled code to the AST interpreter in case a wrong speculation is detected, such as branch miss-speculation. Section 6.6 evaluates the proposed compiler techniques for the R programming language. Finally, Section 6.7 summarises this chapter and the main contributions.

6.1 Motivation

Dynamic programming languages offer high-level abstractions: they are more expressive than static typed languages and allow programmers to write applications with less lines of code. They normally offer a more compact syntax, reducing the boilerplate code and increasing readability and expressiveness. These factors directly affect productivity, allowing fast and iterative software development.

Despite their flexibility, dynamic programming languages are traditionally very slow. For example, a simple operation such as `a + b` performs many runtime checks, such as type checking, checking if any of the operands is `null` or checks if extra memory allocations are needed. Figure 6.1 shows a possible implementation for the plus operator in a dynamic language like R. The left side of the figure shows the operation and its AST representation. The right side shows a Java method that implements the plus operator in the AST interpreter. The method receives two values (left and right) and first performs a type check. If the two values are of the expected type, then the corresponding path (e.g. `executeLong`) is executed.

Implementations of dynamic programming languages make use of VMs that parse



```
public Object add(Object leftValue, Object rightValue) {
  if (leftValue instanceof Long && rightValue instanceof Long) {
    long left = (long) leftValue;
    long right = (long) rightValue;
    return addLong(left, right);
  } else if (leftValue instanceof Double &&
             rightValue instanceof Double) {
    double left = (double) leftValue;
    double right = (double) rightValue;
    return addDouble(left, right);
  } else if …
}
```

Figure 6.1: Example of an AST node implementation for the addition in an interpreted language.

and execute the program with the corresponding specialised typed paths. During the execution, VMs perform allocation and deallocation of objects, perform garbage collection, etc. All of these operations make the overall program execution slow.

To speed-up program execution of dynamically typed languages, VMs compile the most frequently executed sections (called *hotspots*) at runtime, using Just-In-Time (JIT) compilation to CPU machine code. While this approach works well for CPUs, there has been little work for targeting GPUs automatically from dynamically typed programming languages. One of the reasons is the difficulty of implementing a GPU suitable JIT compiler. This task on its own already requires a lot of engineering effort on traditional CPUs. But when running on GPUs, this is even more complicated because of limited programmability features of GPUs compared to CPUs. Currently, GPUs can only execute a subset of C code, with no support for exceptions, traps, recursion, dynamic memory allocation, objects and inheritance.

To understand the complexity of what is being compiled, Figure 6.2 shows the work-flow of an input program executed by the AST interpreter in Java. An input guest language is implemented as a Java program. The interpreter is then compiled using `javac` compiler to Java bytecode. When the interpreter is executed in the JVM, it receives an input program written in the guest language. From the JVM perspective, the input guest program is executed as a normal Java program. When the JVM decides to compile the hotspots, it will compile the hotspot methods of the AST interpreter. For the JIT compiler, the input program is only an ordinary Java bytecode to be compiled and it does not know anything about which language the interpreter implements.

Currently, for application developers, exploiting GPUs from dynamic programming languages is far from trivial since they either have to write the GPU kernels themselves in a low-level language, such as OpenCL or CUDA, or they have to rely on

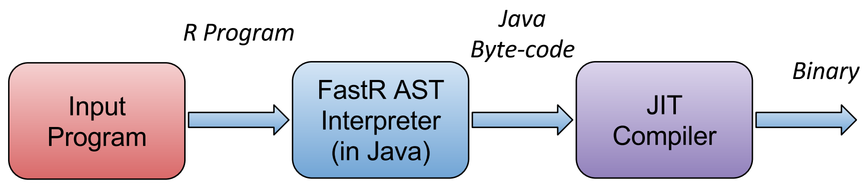


Figure 6.2: Execution work-flow of an input R program written as a guest language on top of the JVM.

third-party GPU accelerated libraries. Ideally, an interpreter for a dynamic programming language would be able to exploit the GPU automatically and transparently. A possible solution seems to be to port the interpreter to the GPU and directly interpret the input program on the GPU. Unfortunately, this naïve solution is not practical since many parts of the interpreter are hard to port to a GPU, such as method dispatch and object representation.

Partial evaluation has emerged as an important technique to improve interpreters' performance on CPUs [Wang 14a, Wurt 13, Wurt 17]. This technique specializes the interpreter to the input program using JIT compilation and profiling information. The compilation unit produced by the partial evaluator is specialized to the data observed during execution and to the hot path in the control-flow as in a trace-based compiler [Gal 06]. As a result, some of the interpreter logic is compiled away, leaving mainly the operations of the input program.

This chapter presents a set of compiler techniques that combines type specialisation, partial evaluation and OpenCL specific specialisation to compile, at runtime, the parallel primitives hotspots of input programs written in interpreted and dynamically typed languages into the GPU with minimal overhead and minimal modifications for language implementers. This chapter shows how an input program is specialised, how the partial evaluator transforms the AST into a compilable unit that is optimised and how OpenCL specialisations are applied before the final OpenCL code generation. The specialised OpenCL program becomes much easier to compile to the GPU since most high-level language constructs and the interpreter overhead are removed away.

This chapter uses R as a use-case to demonstrate the presented compiler techniques to support OpenCL code generation at runtime. The existing FastR interpreter [Kali 14, Stad 16] (R implementation on top of the Truffle language framework) is extended. The R interpreter is slightly modified to represent data parallel operations as parallel nodes in the AST interpreter. When a piece of R code is executed multiple times, the Truffle interpreter specializes the AST using profiling information and transforms it into the Graal Intermediate Representation (IR) using partial evaluation. The Graal IR is then intercepted and a series of passes is applied to simplify the IR as much as possible and the code generator attempts to produce an OpenCL GPU kernel. If the JIT compilation process fails due to unsupported features, the runtime returns to the interpreter automatically and safely using the existing de-optimization process [Holz 92, Wurt 13, Kedl 14].

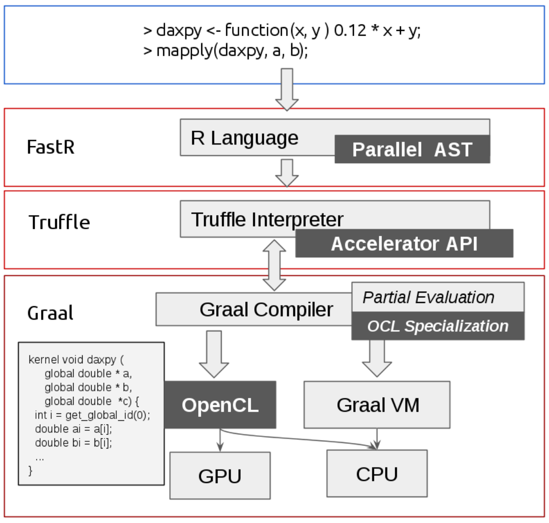
The experimental evaluation in Section 6.6 shows that it is possible to accelerate R programs on the GPU automatically with large gains in performance. This chapter shows that the proposed solution for compiling FastR with OpenCL achieves an average speedup of 150x when using the GPU compared to the runtime of the FastR interpreter on the CPU in peak performance. The proposed approach impressively achieves an average speedup of 57x in a scenario where the VM will perform AST interpreter execution, profiling and JIT compilation, in which OpenCL compilation time and OpenCL device initialization costs are included.

The contributions of this chapter are as follow:

- An OpenCL JIT compiler for the FastR interpreter is presented on top of the Truffle and Graal compiler frameworks. This is the first OpenCL JIT compiler approach for a dynamic programming language where the parallelism is exploited using GPUs with no need for language extensions, and where the code generation and GPU execution are fully automatic.
- A compiler technique to simplify the Graal IR and perform OpenCL specialisations for a more efficient code generation is presented.
- A detailed performance evaluation is presented. It shows how the proposed approach delivers high speedup on a set of R applications running on GPUs compared to GNU-R, FastR and OpenCL C++ implementations.

6.2 OpenCL JIT Compiler for AST Interpreters

This section presents an overview of the proposed compiler approach to compile dynamic programming languages into OpenCL, allowing the execution on heterogeneous devices like GPUs. As already mentioned, the proposed compiler approach is illustrated within the R programming language using the FastR implementation. Therefore, all the examples shown use R but all the techniques are easily portable to other dynamic and high-level languages.



```
> daxpy <- function(x, y ) 0.12 * x + y;
> mapply(daxpy, a, b);
```

```
kernel void daxpy (
    global double * a,
    global double * b,
    global double  *c) {
  int i = get_global_id(0);
  double ai = a[i];
  double bi = b[i];
  ...
}
```

Figure 6.3: System Overview: Starting from R code, the proposed runtime system transparently generates and executes OpenCL code via FastR, which is built on top of Truffle and Graal.

Compiler Approach Overview Figure 6.3 shows an overview of the proposed system compiler infrastructure. Dark grey boxes highlight the proposed extension to the existing FastR, Truffle and Graal compilers. Starting from the R application at the top of the figure, the R implementation (FastR) parses the program and creates an AST. Parallel operations, such as the `mapply`, are handled as special AST nodes. This means that the AST contains nodes that represent parallel operations. First, the AST interpreter executes the operation sequentially and uses profiling information that is needed for the OpenCL compilation in later phases, such as data types and sizes. Then, when the VM detects that the function is hot, it starts the compilation process to execute on the GPU.

As explained in detail in Section 2.4.4, the Truffle specialisation process transforms a generic typed AST to a specialised typed version. When part of the input program is executed enough times, the AST is eventually compiled by Graal via partial evaluation. The OpenCL-specialisation phase removes safety checks from the intermediate representation which are necessary in OpenCL. Finally, an OpenCL kernel is generated – shown in the bottom left corner of Figure 6.3 – from the specialised Graal IR and is executed on a GPU via OpenCL-enabled execution back-end integrated into the Truffle interpreter.

6.2.1 OpenCL JIT Compiler

JIT compilers are complicated pieces of software and machinery integrated into compilation pipelines. A good GPU JIT compiler will also be integrated in those compilation pipelines. The previous section showed an overview of the main compiler components of the proposed compiler approach. This section shows in detail how CPU and GPU compilation work together inside the VM to generate an efficient GPU machine code.

CPU Compiler Work-flow Figure 6.4 shows the work-flow of the FastR execution in combination with the extensions for compiling and running on GPUs. The unmodified FastR implementation performs the steps in white boxes in the following sequence:

1. parsing the R input program;
2. building an AST, similar to the one shown in Figure 2.11.
3. interpreting the nodes in the AST by running their `execute` methods;
4. when the code becomes hot, due to running the program long enough in the interpreter, it is marked for compilation;
5. the partial evaluator produces a control flow graph (CFG);
6. the CFG is optimised and then compiled by Graal to machine code before being executed.

OpenCL Compiler and Execution Engines Rather than applying automatic parallelism, what this solution proposes is to exploit existing primitives in the language that are easily to parallelise with GPUs. Once these primitives are identified in the source code, the FastR interpreter and the compiler perform the following steps, in combination with the ones explained in the CPU compiler work-flow.

1. input and output types analysis;
2. lexical scope analysis;
3. OpenCL specialisation for the IR after partial evaluation only if the input program is selected to execute with OpenCL;
4. OpenCL code generation;

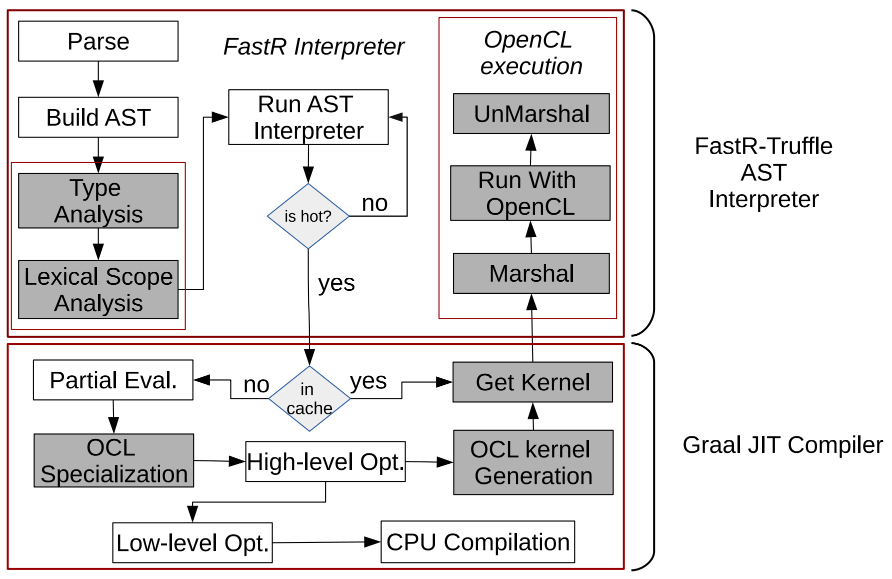


Figure 6.4: JIT Compiler from R interpreter to OpenCL C code. The white squares represent the existing components in FastR and Graal compilers. The grey squares represent the contributions to the existing FastR and Graal compilers.

5. OpenCL execution: this involves the data marshalling (data type transformation between R and OpenCL data layouts), the OpenCL kernel execution and the data un-marshalling.

All of these extensions will be presented and discussed in the next sections. Section 6.2.2 presents how the parallelism is exploited. Section 6.2.3 presents how type inference is performed and Section 6.2.4 shows how lexical scope variables are handled. Section 6.2.5 shows how the `mapply` operation is redefined to support GPU execution and Sections 6.2.6 and 6.2.7 show an overview of how the OpenCL code generation and execution are performed.

6.2.2 Exploiting Parallelism: `mapply` in R

The proposed compiler system for OpenCL JIT compilation neither changes the semantics of the input language nor adds new extensions. Instead, it uses existing language patterns as parallel skeletons.

Most modern programming languages include primitives as operations in their definitions. For example, Python, Perl or Ruby include the `map` primitive, which executes a user function for every element of an input array. This primitive works really well for data parallel applications, as shown in Chapters 4 and 5. In the case of the R language,

```
1 mapply <-function (FUN, ...) {
2   FUN <- match.fun(FUN)
3   if (fastR.oclEnabled())
4     return(.FastR(.NAME="mapplyOCL", FUN, ...))
5   #Default sequential implementation ...
6 }
```

Listing 6.1: Modification of the FastR `mapply` function to support OpenCL execution.

the map primitive corresponds to the function `apply`. This function is widely used in data intensive R programs to apply a function to every element of an input data set. This thesis is focused on the `mapply` variant, which is specialised on array data types. The fact that the order of execution is not specified for the `mapply` primitive makes the parallelisation possible.

One of the constraints when generating machine code, and specially, GPU code, which is based on a different programming model than the one used by the input program, is to preserve the input semantics of the original program.

Parallelisation for the `mapply` One of the main features of dynamic programming languages is that new code can be written or even redefined at runtime. The proposed OpenCL compiler technique exploits this feature to install, at runtime, a redefinition of the `mapply` function that includes OpenCL path enabled. This simple strategy avoids the need to modify the original R source code. When the R interpreter starts, it redefines an updated version for the `mapply` function. Listing 6.1 shows the modification for the `mapply` implementation in FastR. Lines 3-4 check if an `OpenCL` driver is available and create a specialised builtin AST node that is able to generate OpenCL. The new node is called `mapplyOCL`. If there is no OpenCL device available, it will follow the default path with no OpenCL execution.

6.2.3 Type Analysis

Identifying the types of expressions in a dynamically typed programming language is essential during the compilation to a statically typed language, like OpenCL. OpenCL is a statically typed programming language, thus the type analysis must be performed before generating the OpenCL C kernel. To pass the type information to the OpenCL code generator, the FastR interpreter has been extended to analyse, at runtime, the type of variables introduced by the user function that is passed to the `mapply` primitive.

The current implementation supports the most commonly used R data types:

- Vectors of the primitive data types integer, double and logical values.
- R lists: these are vectors that can hold elements of different data types.
- R sequences: these are numeric vectors that contain a predictable sequence of numbers. These are handled specially in FastR to save memory and the OpenCL code generator takes advantage of this optimisation. Section 6.4 explains this process in detail.

Type analysis for input and output values Truffle and FastR automatically perform type profiling to specialise the AST from generic types (the specialisation process is explained in Section 2.4.4). For the OpenCL execution, the following checks are needed.

1. Checking that each input argument (inputs to the lambda and lexical scope variables) has only elements of the same type (monomorphic types). Each input is represented as a primitive array in OpenCL. In OpenCL, elements with different data types can not be mixed within an array. The runtime system creates an OpenCL array for each input array into the input lambda expression.
2. Checking that the input arguments are not `null` references.
3. Checking that the input arguments data types are supported types. The presented compiler technique in this chapter has been implemented for a subset of the R programming language.

To obtain the output data type, FastR is used to execute the R function in the AST interpreter by applying it to the first element of the input data set. This is the same technique that is used for obtaining the output type for Java programs presented in Section 5.3. Based on this execution, the type of the output data is obtained. All of this information is collected and stored as meta-data that is used during the OpenCL code generation.

One important key aspect is that, if Truffle fails to obtain a unique profiled type, if `null` references are detected, or if there is not a supported input or output type, the runtime system falls back to the normal execution and compilation path of FastR. In this manner, the input program is always executed.

```
1 bodies <- runif(n)
2 f <- function(x) {
3     for (i in bodies) {
4         # some computation
5          }
6    ...
7 }
```

Listing 6.2: Sketch of R code that illustrates the use of a lexical scope variable in an inner function.

Type analysis of R lists Lists in R are vectors that can combine elements with different data types. A list in R is represented as a `C-struct` in OpenCL, which is capable of handling different types. When a list is created in the R user code, the modified FastR interpreter for OpenCL creates a `C-struct` to handle the R list in OpenCL. During OpenCL code generation, it creates the corresponding struct.

One of the challenges of R lists is that they are dynamic data structures where the size can change at any point of the execution program. However, the modified FastR interpreter for OpenCL first collects runtime information during the AST interpretation, in which the first iteration is used to obtain the output type as explained above. Moreover, after the partial evaluation and all the compiler optimisation phases, the list to be allocated contains the corresponding size.

6.2.4 Lexical Scope Analysis

The function passed to `mapply` is free to access variables which are defined outside of the function in its lexical scope. Listing 6.2 shows an example in R that illustrates a lexical scope variable (`bodies`), used inside a function. Line 1 creates a vector of random numbers (`bodies`). This array is accessed in the R function via its lexical scope. The loop inside the function `f` in line 3 iterates over the `bodies` array. If an input R program candidate to executed with OpenCL encounters lexical scope variables, the OpenCL component has to, first, detect their type, and second, copy their values to the OpenCL device (e.g. the GPU).

The lexical scope analysis happens for the new AST node of the parallel `mapply` operation. This analysis traverses the AST and, in combination with the stack frame of the `mapply` function, determines which variables have to be copied to the OpenCL device. Then, for each found variable, the type inference is performed.

```
 1 class MApplyOCLNode {
 2    Object execute(RFunction function, RVector input) {
 3       function.compileToOCL();
 4       oclCode = cache.getOCLCode(function)
 5       if (oclCode != null)
 6          return runWithOCL(input, oclCode);
 7       for (int i = 0; i < input.size; i++) {
 8          output.add( function.call( input.at(i) ) );
 9          graph = cache.getGraph(function);
10          if (graph != null) {
11            oclCode = compileForOCL(graph);
12            return runWithOCL(input, oclCode);
13          }
14       }
15       return output;
16    }
17 }
```

Listing 6.3: Run method in the AST interpreter for `OpenCLMApply` node.

The lexical scope variables and their data types are stored and used as meta-data for the OpenCL code generation and execution. During the code generation, the type information is used and at runtime, the type and the value are used to allocate the buffer and to copy the data to the GPU.

6.2.5 AST Interpreter Modification for the Parallel `mapply`

Listing 6.3 shows a sketch of implementation of the execution method of the OpenCL enabled `mapply` node. This method is executed by the FastR interpreter.

Line 3 sets a flag to indicate that the function passed as argument to `mapply` will be compiled to the OpenCL device (e.g. a GPU). The OpenCL compilation process of this function will be discussed in detail in Section 6.3. Next, it checks if the R function has already been compiled to OpenCL code by checking an internal cache, and executes the OpenCL kernel if that is the case (lines 4–6). If the function has not been compiled to OpenCL code yet, it continues executing the input function sequentially in line 8. This executes the function in the Fast R AST interpreter on the CPU. After each iteration of the loop in line 7, the AST interpreted checks if Truffle has performed the partial evaluation in a background thread. If this is the case the cache contains the CFG (line 9). Once the CFG is prepared, it generates an OpenCL kernel in line 11, which is executed on the input vector in the next line.

The shown implementation continues the execution on the CPU in the FastR AST interpreter until Truffle exceeds a compilation threshold and partial evaluation is performed to produce a CFG. This has the advantage that, for small input sizes where the compilation to OpenCL has a significant overhead, it computes the result directly in the AST interpreter. Moreover, it is a minimal modification in the existing FastR AST interpreter that allows language implementers to offload an input function to the GPU.

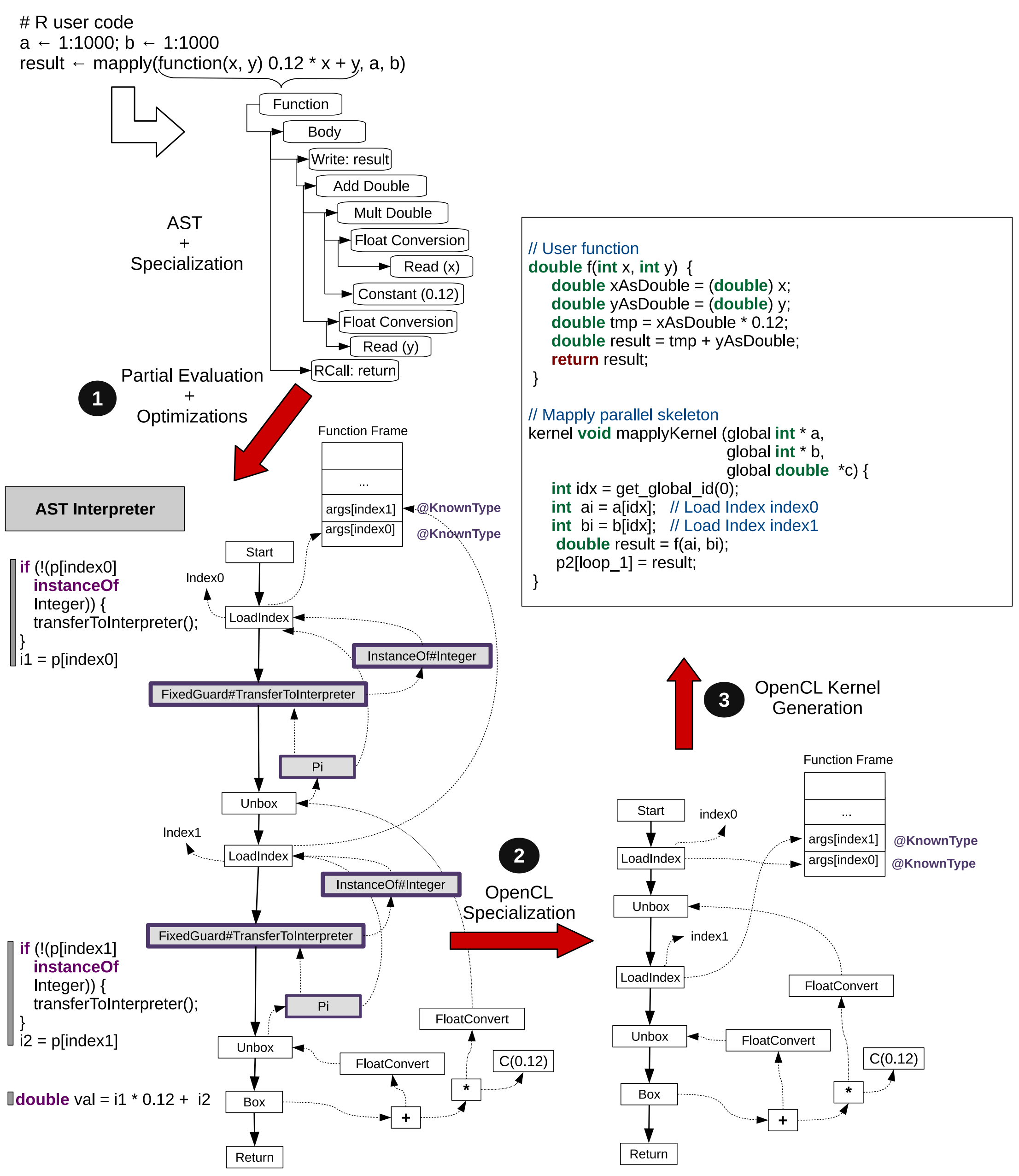
6.2.6 OpenCL Code Generation Overview

The lower half of Figure 6.4 shows an overview of the OpenCL code generation process. The partial evaluation in Graal produces a CFG on which the FastR-OpenCL implementation performs additional specialisations for removing unnecessary checks in OpenCL. After performing multiple compiler-passes for constant folding, inlining, and other common optimisations the compiler generates an OpenCL kernel. The auto-generated OpenCL kernel is then stored in an internal cache to avoid the overhead of generating the same kernel twice for the same input CFG. The OpenCL code generation will be presented and discussed in more details in the next sections.

6.2.7 OpenCL Execution

To execute the generated OpenCL kernel, the underlying VM uses the OpenCL backend presented in Chapter 5. To copy the input data to the GPU, the VM marshals the data from the R data types to OpenCL types, executes the OpenCL program and unmarshals the data.

Section 6.4 discusses how the marshalling can be avoided to improve the overall performance of the input program when running on the GPU.



```
# R user code
a ← 1:1000; b ← 1:1000
result ← mapply(function(x, y) 0.12 * x + y, a, b)
```

```
// User function
double f(int x, int y)  {
    double xAsDouble = (double) x;
    double yAsDouble = (double) y;
    double tmp = xAsDouble * 0.12;
    double result = tmp + yAsDouble;
    return result;
 }

// Mapply parallel skeleton
kernel void mapplyKernel (global int * a,
                                   global int * b,
                                   global double  *c) {
    int idx = get_global_id(0);
    int  ai = a[idx];   // Load Index index0
    int  bi = b[idx];   // Load Index index1
     double result = f(ai, bi);
     p2[loop_1] = result;
 }
```

```
if (!(p[index0]
   instanceOf
   Integer)) {
   transferToInterpreter();
}
i1 = p[index0]
```

```
if (!(p[index1]
   instanceOf
   Integer)) {
   transferToInterpreter();
}
i2 = p[index1]
```

```
double val = i1 * 0.12 +  i2
```

Figure 6.5: OpenCL JIT Compiler from the R interpreter to OpenCL C code via Graal Partial Evaluation.

6.3 OpenCL Code Generation

This section presents the OpenCL code generation process starting from the AST created by FastR into a CFG by Graal via partial evaluation. The CFG is then specialised to OpenCL by removing unnecessary checks. Finally, an OpenCL kernel is generated from this specialised CFG. The code generation is presented and discussed following the example shown in Figure 6.5.

6.3.1 Partial Evaluation and Optimizations

The top left corner of Figure 6.5 shows an input R program that calls the `mapply` function. This code is, therefore, a candidate to execute with OpenCL using the proposed compiler approach. This is the same example shown earlier in Figure 6.3.

During the execution of the input program, FastR first creates an AST and specialises it for the input data it observes during execution. The shown AST in Figure 6.5, is already specialised based on the data types encountered while executing the function as part of the implementation of `mapply` (see Listing 6.3). When the R function is executed often enough by the `mapply` function, the AST of the R function is partially evaluated by Graal into a CFG. To optimise the CFG, a set of common Graal passes are applied, such as function inlining, constant folding, and partial escape analysis [Stad 14b]. Note that the OpenCL JIT compiler for R does not apply any Graal passes which specialize the CFG for the target architecture (e.g. replacing a node with corresponding machine code instructions). The architecture-independent and optimised CFG is shown in the lower left corner of Figure 6.5.

The CFG produced by partial evaluation combines the control flow of the FastR AST interpreter with the original R program. The addition and multiplication nodes at the bottom of the CFG come from the R program. The nodes checking for data type (`InstanceOf#Integer`) come from the FastR AST interpreter, whose Java source code is shown on the left side.

In the example, the FastR AST interpreter has been specialised for the **`int`** data type. This is because the input is a sequence of integer numbers, therefore the FastR interpreter specialises the type to an internal `RIntVector` that stores an explicit array of integers. The Java code on the left side shows the guarding code that checks if the optimization assumption is broken and deoptimizes if that is the case.

6.3.2 OpenCL Specialization

When the input R function is executed on a GPU, the compiler and the runtime system guarantee that the input data must have the proper data type, `Integer` in this case. That is true, because the runtime system marshals the data prior to sending it to the GPU where it can ensure that the data types of all elements must match. Therefore, those checks can be safely removed from the AST interpreter and, therefore, from the CFG.

The FastR interpreter, Truffle and Graal compiler have been extended to provide a generic mechanism for eliminating such AST interpreter overheads and, thus, specialize the CFG for OpenCL execution. Truffle defines a set of *compiler directives* which are annotations conveying information useful for specialising and optimising the CFG. One example is the existing `@CompilationFinal` directive shown in Figure 2.11 which specifies that a value will not change anymore and, therefore, can be assumed constant by the Graal JIT compiler.

In this thesis, new compiler directives for OpenCL are proposed:

- `@NotNull`: specifies that the annotated variable is guaranteed not to be **`null`**.
- `@KnownType`: specifies that the type of the annotated variable is known because of the explicit data transfer from the OpenCL host the OpenCL device.
- `@ArrayComplete`: specifies that the annotated array is guaranteed to not contain `NA` values. This annotation is specific for the R programming language, however, other Truffle languages can benefit from it if they have similar semantics.

The language implementer uses these annotations in the AST interpreter to inform the OpenCL JIT compiler about where the extra checks are. In the CFG, the arguments are part of the `FunctionFrame` shown as a table in Figure 6.5.

For functions marked as being compiled to OpenCL, as shown in Listing 6.3, an OpenCL-specific compiler pass is performed. This compiler-pass traverses the CFG and removes nodes by exploiting the additional information given by the OpenCL specific compiler directives. The highlighted `FixedGuard#TransferToInterpreter`, `Pi` and `InstanceOf#Integer` nodes, which dynamically check that the given parameter is of type `Integer` and convey this additional type information to the compiler using the `Pi` node, will be removed based on the `@KnownType` directive.

Using this strategy, the OpenCL JIT compiler only removes checks where it can be sure that this is legal and safe. Otherwise, it will keep the checks in the auto-generated OpenCL C code. These checks are used for deoptimizations because of a miss speculation during the profiling information in the AST interpretation. Graal is an aggressive compiler and it only generates code for the branches that have been profiled. If there is no profiled branch, Graal inserts a guard to deoptimise. Section 6.5 presents deoptimizations on GPUs using the OpenCL JIT compiler approach presented in this thesis.

6.3.3 OpenCL Kernel Generation

After specialising the CFG for OpenCL, most of the interpreter logic for handling exceptional cases on the CPU are gone. The remaining CFG, shown in the bottom right corner of Figure 6.5, has still the nodes corresponding to the original R program. The function `f` shown in the top right is generated by traversing the CFG and emitting an OpenCL snippet for each node using the compiler infrastructure presented in Chapter 5. The remaining OpenCL kernel is generic for the `mapply` function. The two input vectors (`a` and `b`) are passed as arguments to the `mapply` kernel where the function `f` is called with two elements of the input user code.

6.3.4 GPU Code Cache

Once the OpenCL kernel has been generated, the runtime system inserts the OpenCL compiled kernel into an internal cache. This cache is separated from the JVM cache and it is managed independently from the CPU cache binary. If the compiled function is executed again, the runtime first checks if there is an existing OpenCL kernel already compiled in the cache and executes it directly from the cache.

6.4 Data Management Optimisations

This section discusses two important optimizations implemented for efficient data management between R and GPUs.

Avoiding of marshalling for R vectors Vectors are the essential data structure in R to store collections of values which have the same data type. FastR specializes its implementation of R vectors based on the data type of the elements. Instead of storing an array of objects, FastR uses directly a primitive array for storing the values of a specialised R vector. For example, a **`double`** array for an R vector of double values. The underlying implementation for OpenCL takes advantage of this specialisation by FastR, as fortunately, primitive arrays are already in the byte format required for the GPU and no extra marshalling step to translate the data managed by FastR into a format accessible by OpenCL is required.

To expose this primitive array in the R vector implementation to OpenCL, a new modification of the presented `PArray` data structure in Chapter 5 is implemented. The `PArray` is a data structure that stores an input array in the Java format object layout

```r
x <- 1:size; y <- 1:size   # R sequences
mapply(function(x, y) 0.12 * x + y, a, b)
```

Listing 6.4: R example to compute Daxpy with input sequences of integers

into the OpenCL layout. This strategy avoids the data type transformation between R and OpenCL C. Then, the `PArray` implementation passes the R primitive array to the OpenCL implementation which copies it into the GPU. The evaluation section of this chapter shows the impact of this optimization for the overall runtime of the R user application.

Optimization of R sequences A sequence in R represents all values between a *start* and *end* values which can be reached with a given *step* size. In FastR sequences are handled specially: instead of storing the elements in memory, they are generated on-demand by evaluating the formula: *start* + *stride* * *index*. This is especially beneficial in the context of OpenCL because the formula is inlined in the OpenCL C code whenever an R function takes a sequence as its input argument. This saves the costly data transfer to the GPU.

In Listing 6.4, the `x` and `y` variables are sequences ranging from 1 to a given `size`. Listing 6.5 shows the OpenCL code corresponding to the R sequence specification `1:size`. The OpenCL thread id in line 4 is used as the *index* in the computation of the formula listed in lines 5-6.

This technique has two clear advantages:

1. OpenCL buffers and data transfer between the CPU and GPU are completely avoided. The OpenCL-FastR runtime only passes the `start` and the `stride` values which are independent from the input array size;

2. Accesses to the slow OpenCL global memory into the input vector are also avoided. This has a clear positive effect on the overall performance as the evaluation section will show.

```
kernel void mainKernel (global int * p0,
                        global int * p1,
                        global double  *p2) {
    int idx = get_global_id(0);
    int temp1 = 1 + 1 * (idx);
    int temp2 = 1 + 1 * (idx);
    Tuple_int_int s;
    s._1 = temp1;
    s._2 = temp2;
    double result= callRoot(s);
    p1[idx] = result;
}
```

Listing 6.5: OpenCL code snippet for R sequences

```
1 mapply(input, function(x) {
2   if (x < 1) return(0)
3   else return(1) })
```

Listing 6.6: R function with a input depending branch

6.5 Handling of Changes in Program Behaviour

As shown earlier, Truffle uses specialisation to optimise the AST during execution. The specialisation in the AST interpreter is based on profiling information gathered during program execution. Graal optimistically speculates based on the assumption that the profiling information of past executions is representative for future executions. Guards are introduced to handle changes in the program behaviour when this assumption is broken and a deoptimization is performed.

Deoptimizations in OpenCL A simple example is now used to illustrate how the deoptimization process works with OpenCL. The execution work-flow for the R code showed in the Listing 6.6 is illustrated in Figure 6.6. Listing 6.6 shows an R program which applies a function to an `input` vector. The function has a branch in line 2 that depends on the input `x`. If `x` is less than `1`, the function returns `0`, otherwise `1`.

Let us assume that this function is executed on a large input array where the first

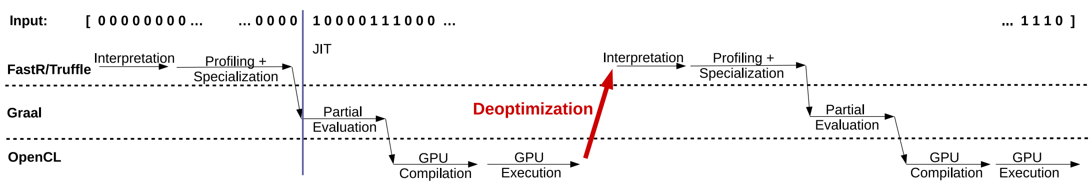


Figure 6.6: Example of execution work-flow for an input vector that executes the R code showed in Listing 6.6. The R program is specialised, and aggressively compiled to the GPU with branch speculation. Then a deoptimisation is captured at runtime due to a miss speculation. Then the program is re-profiled and compiled to the GPU again, and, finally, successfully executed on the GPU with no deoptimisations.

```
1 double f(double x,global int* deoptFlag) {
2   bool cond = x < 1.0;
3   if(!cond) *deoptFlag = get_global_id(0);
4   return 0.0;
5 }
```

Listing 6.7: OpenCL C code generated for the R program in Listing 6.6.

1000 elements have the value `0`. As shown at the top left corner of Figure 6.6, FastR will start interpreting the code and profile the execution by collecting statistics on which branches are taken in the execution. For the execution of the first elements, the profiler will not register an execution of the **`else`** branch. Based on this profiling information, the AST is specialised by FastR and after sufficient iterations the AST is handed to Graal for compilation via partial evaluation, as shown in Figure 6.6. The partial evaluation speculatively removes the computation in the **`else`** branch based on the profiling information. A guard checks that in case the **`else`** branch is hit, in which the execution is returned to the interpreted code.

As a consequence of this behaviour, the OpenCL kernel generator operates on a CFG where the **`else`** branch is no longer present and generates the code shown in Listing 6.7. This code generator generates an extra variable where the OpenCL thread-id for deoptimising is stored. This buffer is initialised, at runtime, to -1 (because there can not be any OpenCL thread with negative values).

The code unconditionally returns `0` which corresponds to the **`true`** branch of the original program. A guard checking for the optimization assumption is generated in line 3 where a global flag is set to indicate the deoptimization.

```
double f(double x,global int* deoptFlag) {
  bool cond = x < 1.0;
  if (cond) return (0.0);
  else return (1.0);
}
```

Listing 6.8: OpenCL C code generated for the R program in Listing 6.6 after re-profiling

GPUs are a parallel hardware with no support for raising exceptions. Therefore, the whole computation has to be performed before checking if there was an exception in some point of the OpenCL kernel execution. When it is finished, the runtime system checks if the `deopt` flag was raised by any thread. If this is not the case and the flag is still set to its initial value of -1, the speculation was correct and the computed result is returned. Otherwise, the runtime performs a deoptimization as indicated with the red arrow in Figure 6.6. This invalidates the generated OpenCL kernel and the control is transferred back to the FastR AST interpreter, which continues interpreting the program and collects more profiling information. By storing just the OpenCL thread-id, the runtime can execute in the AST interpreter with the OpenCL thread-id that provoked that exception (each thread maps one element from the input data set).

The result is `deoptFlag` is undefined because multiple threads can write to the same location. However, the flag variable is write-only with no other math operation involved. If the data is aligned, the transaction is performed atomically. As the OpenCL standard [OpenCL 09] specifies, data is always aligned to the size of its data type in bytes. Therefore, it is not possible to have a junk value in this flag because the transaction of 4 bytes (`int` value) is, in fact, performed atomically.

Concerning the value that the `deoptFlag` can have at the end of the kernel execution, this is undefined. However, it is enough for the runtime to obtain one of the thread identifiers (thread-id) that provoked the deoptimization. Writing the thread identifier into the flag has the advantage, that the runtime system can force the FastR AST interpreter to interpret the program with the input data of least one particular thread that provoked the deoptimization. When the FastR AST interpreter reaches its compilation threshold again, (as indicated in Figure 6.6) the code is recompiled to OpenCL via partial evaluation with the updated profiling information.

Benchmark	Input (MB)	Output (MB)
Daxpy	128	64
Black-Scholes	8	16
NBody	5	3
DFT	0.156	0.156
Mandelbrot	16	8
Kmeans	64	16
Hilbert Matrix (HM)	128	128
Spectral Norm (SN)	0.256	0.256

Table 6.1: Benchmarks and default data sizes used to evaluate the FastR-OCL implementation.

The kernel shown in Listing 6.8 is generated. The additional profiling information has helped generalising the AST and both branches are now visible to the OpenCL code generator. This technique is a very simple strategy to handle deoptimizations with minimal overhead in the OpenCL kernel.

More advanced techniques, such as storing a flag per thread to indicate which one to deoptimise, would be possible. However, this has the main disadvantage of wasting more memory on the GPU global memory.

6.6 Evaluation

This section presents a performance evaluation of the OpenCL JIT compiler approach using the R programming language. It first describes the machine and environment setup and then it presents and discusses the performance results.

6.6.1 Experimental Setup

The OpenCL JIT compiler for the R language has been evaluated using two different GPUs; an AMD Radeon R9 295X2 with 8GB memory and a GeForce GTX TITAN Black with 6GB memory. The OpenCL drivers correspond with the AMD 1598.5 and the Nvidia 367.35. The CPU used in the experiments is an Intel Core i7 4770K @ 3.50GHz with 16GB of DDR3 RAM.

This section provides a comparison between FastR, GNU R, OpenCL C++ and the proposed compiler R interpreter with GPU enabled version of FastR (called *FastR-*

OCL). It is hard to make a fair comparison between GNU R and FastR because they have a different VM and JIT compilers. The GNU R version is the 3.3.1 with the *enableJIT* option set to level 3 which enables the JIT compiler. Both FastR and FastR-OCL are running on top of the Graal 0.9 Java VM (Virtual Machine). The OpenCL JIT code generator is built on top of the Graal VM JIT compiler which is exposed via a Java API.

6.6.2 Measurements

Each benchmark has been executed 10 times and the median runtime has been reported. This means that the peak performance of the overall execution is reported. The Java heap memory is set to 12GB to avoid the execution of the garbage collection in the middle of an execution. This avoids the introduction of noise in the measurements. For FastR and FastR-OCL, time measurements are performed using a custom built-in, which internally calls `System.nanotime()`. For GNU R, the time has been measured by calling the `proc.time()` function. All times presented in this section are measured from R and include GPU related overheads such as data management. For the speedup comparison the geometric mean is reported as average of all executions.

6.6.3 Benchmarks

The benchmarks evaluated in this chapter are the ones presented in Chapter 4.6 and evaluated in Section 5.6 on the GPU for Java applications and another three benchmarks from the R benchmark suite. All the benchmarks were rewritten in R following data parallel benchmark implementations from Rodinia [Che 09], the AMD OpenCL SDK [SDK 16] and the programming language benchmarks game[1]. The benchmarks were selected based on typical and representative applications for data and compute intensive applications from different domains: `Daxpy`, `Black-scholes`, `Mandelbrot`, `DFT`, `NBody`, `Kmeans`, `Hilbert Matrix` and `Spectral Norm`.

Table 6.1 shows the data sizes used for the execution of the benchmarks of the first two experiments. The sizes are chosen to be large enough to lead to sufficiently long runtimes in FastR but are fairly small for massively data parallel GPUs. This section will show that, for even moderate data sizes for GPUs, large speedups are obtained compared to FastR.

[1]`http://benchmarksgame.alioth.debian.org/`

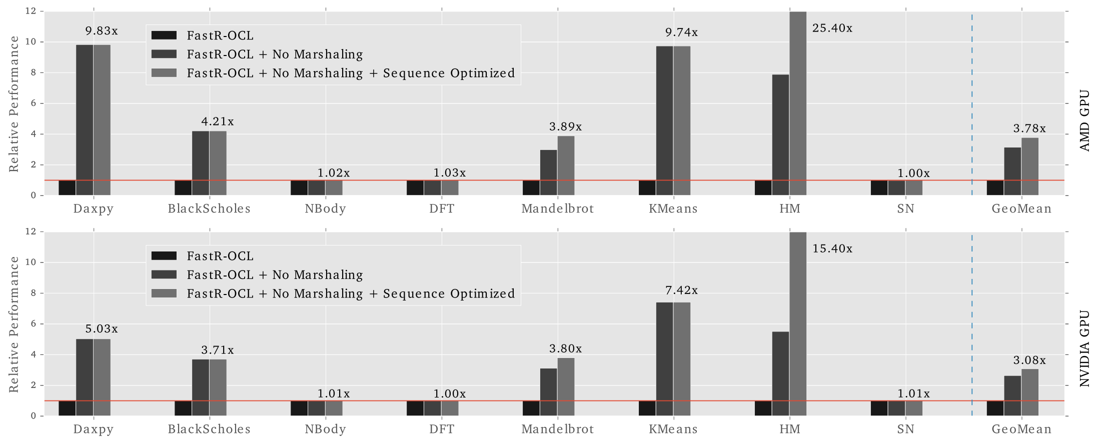


Figure 6.7: Performance impact of data management optimizations in FastR-OCL.

6.6.4 Impact of Data Management Optimizations

This section analyses the effect of the optimisations presented in Section 6.4. Figure 6.7 shows relative performance of three versions of FastR-OCL implementation: 1) the leftmost bar shows the naive version which performs marshalling for R vectors and sequences; 2) the middle bar shows the optimised version avoiding marshalling for R vectors and sequences, but performs data transfers for R sequences; 3) the rightmost bar shows the most optimised version avoiding marshalling for R vectors and also avoiding data transfers for R sequences. The graph shows the performance for AMD at the top and NVIDIA at the bottom.

The benefits of the data optimization are dependent on the compute intensity of the benchmark. For compute intensive benchmarks, such as `Nbody`, `DFT`, and `SN` the optimisations have a minor effect, as the vast majority of execution time is spent in the computation on the GPU. For all other benchmarks, the benefits are more significant with runtime improvements of up to $25.4\times$ for the `HM` benchmark. Since the `HM` program takes two R sequences as input, it is possible, through the optimisation sequence, to remove most of the data transfer, since an RSequence can be simply represented by three numbers: start, stride and end. Overall, the average improvement of the FastR with OpenCL data optimizations is $3.78\times$ on AMD and $3.08\times$ on Nvidia.

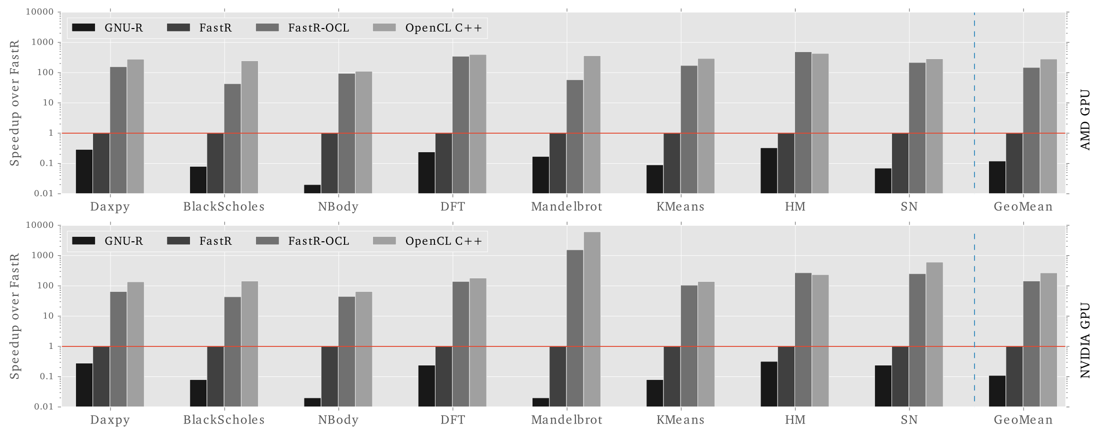


Figure 6.8: Speedup of FastR-OCL executing on AMD and Nvidia GPUs compared to GNU-R, FastR, and OpenCL C++.

6.6.5 Performance Comparison

Figure 6.8 shows a performance comparison for each benchmark with GNU R, FastR, FastR-OCL implementation, and a native hand-optimized OpenCL implementation of each benchmark in C++. The bars show speedup over the sequential execution with FastR. Due to the large speedups achieved by the GPUs, the y-axis is in logarithmic scale. The input R programs used in this evaluation are exactly the same for GNU R, FastR, and for the FastR-OCL implementation.

Figure 6.8 shows that FastR is 10 to 100× times faster than GNU R for these data intensive benchmarks, confirming prior results [Stad 16, Wurt 17]. When using the AMD GPU, FastR-OCL approach is 150× times faster than FastR on average and 1000× faster than GNU R. Using the Nvidia GPU with FastR-OCL is, on average, 130× faster than FastR. For some benchmarks, such as `Mandelbrot`, FastR-OCL achieves a speedup of more than 1300× compared to FastR by using the Nvidia GPU.

The last set of bars shows the performance achieved by a highly tuned manually written native C++ OpenCL implementation. For the `nbody`, `dft`, `HM` and `SN` benchmarks, FastR-OCL achieves very similar performance to the native OpenCL implementation. For benchmarks such as `Mandelbrot`, the C++ OpenCL implementation is clearly faster due to the fact that it exploits parallelism in multiple dimensions where the Marawacc code generator only exploits a single dimension of the OpenCL thread iteration space. On average, the FastR-OCL implementation is about 1.8× slower than the native OpenCL implementation. Although there is room for improvement, this is fully achieved automatically, starting from a program written in the R dynamic interpreted language.

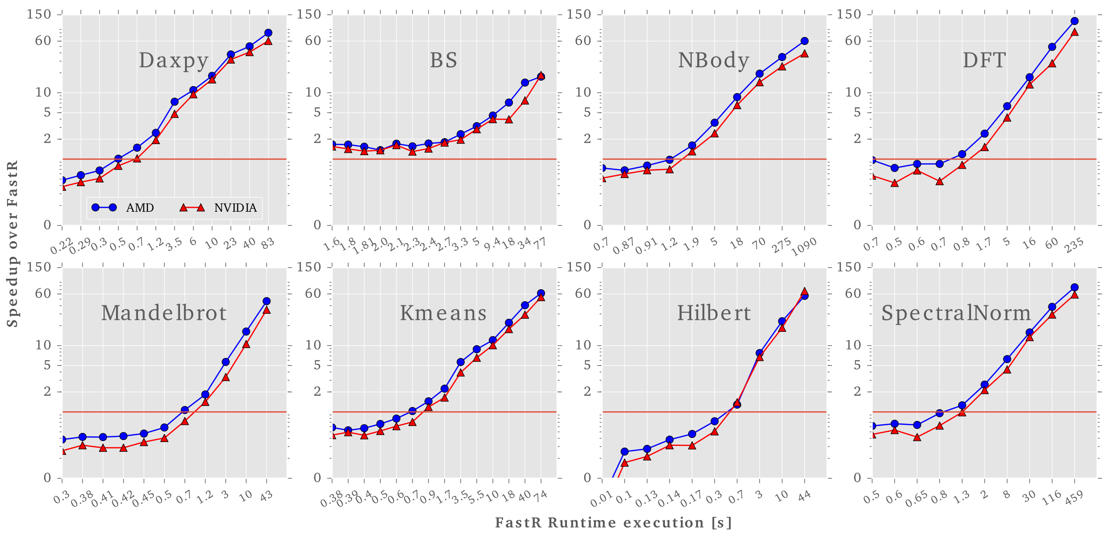


Figure 6.9: Speedup of FastR-OCL over FastR executed with a cold JIT compiler with increasing input sizes from left to right. The x-axis labels show the FastR runtime while the y-axis shows the speedup of the proposed FastR-OCL implementation. Even R programs running only for a few seconds achieve large speedups.

6.6.6 Performance of Cold Runs

The experiment evaluations presented in 6.6.4 and 6.6.5 have been measured as a median of multiple runs inside the same R session running on top of the Java VM. Therefore, the VM had time to warm up and perform the JIT compilation before reporting a measurement.

A more typical end user scenario is to execute the R program only once with a fresh JIT compiler state. This scenario means that the VM starts and the start timer is set in the first line before running in the AST interpreter. Then, all the times for interpretation and JIT compilation are included. The stop timer is located at the end of the execution (end-to-end). This process has been repeated 10 times and the median value is reported to account for noise.

Figure 6.9 shows a performance comparison of FastR and FastR-OCL measured using a cold JIT compiler. For each benchmark, the input size is increased in power of two, from left to right and the x-axis labels show the FastR execution time. The y-axis shows the speedup achieved of FastR-OCL over FastR for the AMD (blue circle) and Nvidia (red triangle) GPU.

Across all benchmarks, it can be seen that even the case of a cold JIT compiler with small data sizes, which executes for only a few seconds in FastR, is sufficient for achieving a speedup using the GPU via FastR-OCL. For example, using the GPU be-

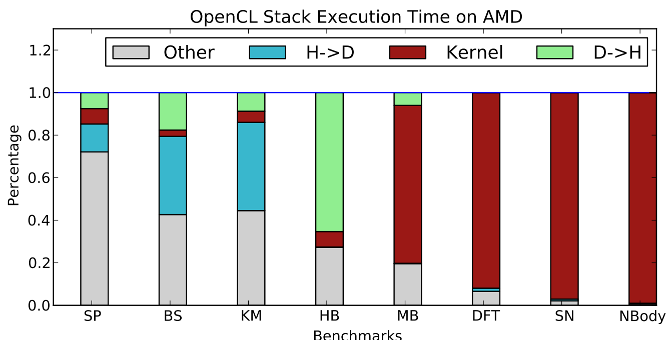


Figure 6.10: Breakdown of the OpenCL execution run-times for 8 R benchmarks executed on an AMD ATI GPU.

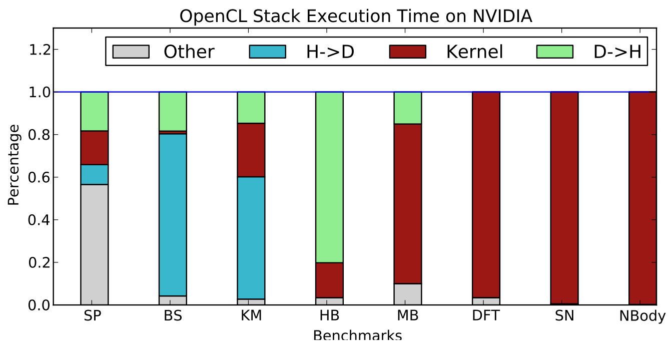


Figure 6.11: Breakdown of the OpenCL execution run-times for 8 R benchmarks executed on a Nvidia Gforce GPU.

comes beneficial for the simple daxpy benchmark for input sizes where the computing process takes longer than 0.7 seconds in FastR. For an input size where FastR runs for 83 seconds, FastR-OCL obtains speedups of more than 57$\times$ for both GPUs. Overall it can be observed that despite the AST specialisation, partial evaluation, OpenCL kernel generation and compilation, large speedups are obtained even for R programs which execute only for a few seconds.

6.6.7 Breakdown of OpenCL Execution Times

Figures 6.10 and 6.11 show a breakdown of the peak performance OpenCL run-times for all benchmarks on the AMD and NVIDIA GPUs. The total runtime is broken down into four parts: copying the data to the GPU ($H \rightarrow D$), the kernel execution, copying the data from the GPU ($D \rightarrow H$) and remaining time (other) that includes the runtime of the interpreter. The data sizes used for these experiments are shown in Table 6.1.

Kernel Execution The benchmarks that achieve the highest speedups with FastR-OCL on a GPU, are those where the predominant part is the OpenCL kernel execution. This is the case for NBody, DFT and Spectral Norm, where the OpenCL computation takes up to 90-99% of the overall runtime. For the rest of the benchmarks, the kernel takes between 3-15%. In these cases, an efficient data management is crucial to achieve good performance.

Data Transfers Figures 6.10 and 6.11 show clearly the benefits of the FastR-OCL data transfer optimizations for the R sequences in OpenCL. This is the case of DFT, Mandelbrot, Hilbert and Spectral Norm, where copying the data to the GPU takes less than one per cent. For some benchmarks the data transfer time takes up to 75% of the overall execution time. However, as shown in Figure 6.9, even with the data transfers to and from the GPU, R programs can benefit with large speedups.

6.6.8 Compilation Time

Table 6.2 shows the compilation time for each benchmark. Partial evaluation and optimizations performed by Graal take significantly longer than the OpenCL kernel generation itself which takes only about 11ms. However, the compilation of the OpenCL kernel by the GPU driver can take up to 250ms. The table shows that the overhead for OpenCL kernel generation (second column) is less than the compilation for the AMD platform and similar to the compilation for the Nvidia platform (third and fourth columns). This low overhead also explains why using the GPU is already beneficial for small input sizes, as shown before.

The actual OpenCL GPU compilation time shows that further improvements are possible by directly targeting the GPU instruction set and bypassing the OpenCL compiler. As future work, the use of OpenCL SPIR (Standard Portable Intermediate Representation, see 3.1.3) instead of plain OpenCL source code will be investigated in order to decrease, even more, the compilation overhead. However, as the results have shown, large speedups are already achievable, even when relying on the OpenCL compiler.

6.6.9 Evaluation Summary

This performance evaluation has shown that accelerating R programs with FastR-OCL is not just feasible but highly beneficial for data parallel applications. Exploiting the parallel power of the GPU leads to large speedups. The proposed data optimizations to

Time in ms				
Benchmark	**Partial Evaluation + Optimisations**	**Kernel Geneneration**	**Compilation AMD**	**Compilation Nvidia**
Daxpy	106.27	7.27	29.19	81.32
Black-scholes	94.54	12.70	55.79	180.20
NBbody	93.59	11.35	47.04	109.82
DFT	95.09	20.12	61.56	250.86
Mandelbrot	114.31	9.37	34.70	102.78
K-Means	113.54	9.62	41.33	93.43
Hilbert	105.44	7.74	27.82	95.67
Spectral N.	146.57	11.86	87.03	219.53
Mean	107	11	48	142

Table 6.2: Time (in milliseconds) for different phase of the GPU code generation process: Partial evaluation and optimizations, OpenCL kernel generation and compilation by the GPU driver.

avoid marshalling for R vectors and data transfers altogether for R sequences increase performance by up to 25×. This allows to achieve speedups ranging from 43× up to more than 150× when compared to the sequential execution in FastR. Even when executing the R program only a single time on a cold JIT compiler, large speedups can be obtained. It has also shown that using the GPU is beneficial even for short running R programs.

6.7 Summary

This chapter has presented a compiler technique to transparently and automatically offload computations from dynamically typed programming languages to GPUs. This technique has been implemented for the R programming language as a modification of the FastR interpreter, which is built on top of Truffle and Graal. To the best of our knowledge, this chapter presents the first OpenCL JIT compiler for the R programming language that automatically compiles a subset of the input program without any extension in the source code.

This chapter has discussed about the challenges of generating OpenCL code from dynamic languages. The proposed solution combines type specialisation in the AST, partial evaluation and OpenCL-specific specialisations to simplify the intermediate representation of the input program before generating the OpenCL code. Part of the proposed technique includes the use of specific compiler directives to convey optimization information for avoiding unnecessary checks on GPUs.

The proposed solution includes data runtime optimizations for avoiding marshalling of R vectors and an optimized handling of R sequences types, which even avoid costly data transfers to GPUs. This approach is also able to handle cases where a specialised AST has to be deoptimized and generalised during execution. On average this compiler approach is $150\times$ faster than FastR on a range of data intensive benchmarks and only $1.8\times$ slower compared to manually written OpenCL code. Speedups of $57\times$ can be achieved even for small R programs when using a cold JIT compiler.

Part III

CONCLUSIONS

Chapter 7

Conclusions

This thesis has presented three main contributions that help to reduce the gap of programmability between high-level languages and GPUs. They are a) a Java Application Programming Interface for data parallelism and array programming, b) an OpenCL JIT compiler that automatically translates Java bytecode into OpenCL and with a runtime system that executes and optimises GPU programs within Java and c) a set of compiler techniques to automatically compile dynamically typed languages to OpenCL.

This thesis has presented these techniques for interpreted programming languages, in particular for the Java and R programming languages on top of the Java Virtual Machine using the Graal compiler. To make the GPU compilation and execution feasible for dynamically typed languages, the proposed solution extends existing compiler techniques that uniquely combine specialisation and partial evaluation for OpenCL, in which language interpreters are specialised to the input program. This allows to compile and execute user applications implemented in dynamically typed languages to GPUs.

This chapter reviews the problem that this thesis has addressed, summarises the contribution and presents the limitations and future work. Section 7.1 summarises the problem that has been investigated in this work. The main contributions of this thesis are summarised in Section 7.2. Section 7.3 shows a critical analysis of these contributions and presents solutions to overcome some of the constraints. Finally, Section 7.4 shows possible future directions of this work.

7.1 Problem

Graphics Processor Units (GPUs) are powerful hardware that is widely used in industry and academia. They have unlocked the possibility for many scientists and developers of exploiting parallel applications and process a large amount of data in shorter time. GPUs are programmed using low-level programming languages, such as CUDA and OpenCL. These programming languages require a deep understanding of hardware and parallel computing.

However, many of the current GPU users are not experts. They usually develop GPU applications in higher programming interfaces than CUDA and OpenCL using interpreted and dynamic programming languages. Current parallel programming interfaces either provide an API with a fixed number of operations to execute on GPUs or provide a library to directly program GPU applications within interpreted languages (e.g. via wrappers).

Little work has been done on automatically exploiting parallelism on GPUs from interpreted programming languages. To overcome these limitations, this thesis has presented a combination of compiler techniques, together with an API for programming parallel and heterogeneous applications from Java and R programming languages. This thesis has focused on the Java programming language, as the primary representative of interpreted languages, and the R programming language, as the main representative one of dynamically typed languages.

7.2 Contributions

This section summarises the main contributions presented in this thesis in Chapters 4, 5 and 6. Chapter 4 has shown JPAI, a new API for array and heterogeneous programming in Java. Chapter 5 has presented an OpenCL JIT compiler for the JVM, which compiles Java bytecode into OpenCL C, and a runtime that optimises data transformations between Java and OpenCL through the proposed `PArray` data structure, a custom array type for avoiding the data transformation between Java and OpenCL. Chapter 6 has presented a set of compiler techniques to compile, at runtime, programs that are written with interpreted and dynamically typed programming languages into OpenCL C. It combines techniques such as type specialisation, partial evaluation, language annotations and data optimisations to efficiently compile a high-level input program into OpenCL.

7.2.1 Java Parallel Array Interface

Chapter 4 has presented JPAI, a new API for array and heterogeneous programming in Java. In contrast to the related works for GPU programming within Java, JPAI relies on composability and reusability. These allow programmers to compose multiple parallel operations and reuse the same computation multiple times independently of the input data set. JPAI is focused on data parallelism and array programming.

JPAI also integrates a runtime system that automatically executes Java applications written with JPAI on a multi-core CPU or a GPU. For a multi-core execution, the JPAI runtime splits the iteration space into smaller chunks and processes each of them in independent Java threads. For the GPU execution, JPAI relies on another component, Marawacc, that compiles, at runtime, the user computation into OpenCL and executes the generated code on the GPU.

User applications written with JPAI are implemented independently of the underlying parallel hardware. JPAI transparently exposes parallelism to programmers through Java threads or via OpenCL. However, independently if the application is executed on a multi-core CPU or a GPU, the input source code remains the same.

Chapter 4 has also presented a set of scientific applications written in Java using the JPAI programming interface. In addition, it has evaluated the performance of JPAI on a multi-core CPU employing Java threads. The results have shown that JPAI obtains speedups of up to 5x using four cores with hyper-threading over Java sequential code.

7.2.2 OpenCL JIT Compiler and a Runtime System for Java

Chapter 5 has presented Marawacc, a framework for heterogeneous computing in Java. Marawacc is composed of an OpenCL JIT compiler that automatically translates Java bytecode to OpenCL, and a runtime system that orchestrates and optimises the GPU execution from Java. Marawacc uses JPAI to exploit parallel applications implemented in Java and execute them on the GPU. Marawacc uses the JPAI parallel skeletons and the user computation to offload, at runtime, the corresponding Java code to the GPU automatically.

The OpenCL JIT compiler transforms the user defined function and the parallel skeleton into the Graal IR (Graal intermediate representation in a control flow graph form). Then, this OpenCL JIT compiler applies a set of compiler transformations in the CFG to optimise the code. Once the CFG is optimised, the OpenCL JIT compiler generates the corresponding OpenCL code by traversing each of the IR nodes and generating the equivalent OpenCL C code. The Marawacc OpenCL JIT compiler is integrated as a plugin for the Graal compiler. Marawacc can benefit from the existing powerful compiler optimisations available in Graal, such as inlining, constant propagation, node replacement and partial escape analysis. Marawacc's approach is, in fact, different from the state-of-the-art works, in which the OpenCL translation is performed directly from the Java bytecode, except for the Tornado compiler framework, which uses a similar approach to Marawacc.

Instead, the OpenCL JIT compiler is integrated with the Graal compilation pipeline and it only uses the high-level intermediate representation, which allows a faster implementation of a JIT compiler compared to the traditional JVM compilers.

Chapter 5 has also presented a runtime system that orchestrates and optimises the GPU execution. The Marawacc runtime exposes the `PArray` data structure to programmers to avoid the data transformation between Java and OpenCL (called *marshalling*). This allows programmers to increase the overall performance of their applications when running on GPUs.

Moreover, a detailed performance evaluation of Marawacc has been presented. Marawacc has been compared to the state-of-the-art works of Aparapi, JOCL and native OpenCL. It shows that, in average, Marawacc is 23% slower than the native implementation and 55% faster than similar approaches such as Aparapi.

7.2.3 GPU JIT Compilation for Dynamic Programming Languages

Chapter 6 has presented a set of compiler techniques to dynamically offload, at runtime, part of a program written in an interpreted and dynamic language to the GPU via OpenCL. The R programming language has been selected as a use-case. Providing a GPU JIT compiler for dynamic programming languages is challenging. The main challenge is the data type inference of dynamically typed languages. One of the possible solutions would be to implement all the logic to infer types on the GPU. However, this solution can easily degrade the overall performance. To overcome this problem, the proposed solution leverages the existing compiler techniques of specialisation and partial evaluation with OpenCL specialisation and compilation in order to efficiently execute an R program on GPUs.

The proposed solution extends many parts of the existing R interpreter and compiler. The AST interpreter has been extended to include profiling information that will be propagated to the partial evaluator for OpenCL code generation. It has also introduced a set of annotations in the language implementation level to report more information to the compiler, allowing to simplify the IR as much as possible to reduce the interpreted overhead. The runtime system optimises the data transformation between R and OpenCL by extending the `PArray` data structure for R vector types, allowing to remove the *marshal* and *unmarshal*. The runtime system also applies optimisations for some of the R types, such as R-sequences of numbers where the data can be computed directly on the OpenCL kernel, saving GPU buffer allocation, data transfers and accesses to the GPU global memory. Moreover, Chapter 6 has also shown how to dynamically perform deoptimisations in the OpenCL code using a straightforward technique to store the OpenCL thread-id (*work-item*) that provoked the deoptimisation.

Finally, a performance evaluation of FastR with GPU compilation for a set of R benchmarks running on GPUs has been presented. The FastR-GPU approach has been compared to the GNU-R compiler, FastR and the manually optimised native OpenCL on two different GPU platforms. This evaluation has shown that, by compiling the R application at runtime and executing on a GPU, it is possible to achieve speedups of up to 150x over FastR and 1.8x slowdown compared native OpenCL.

7.3 Critical Analysis

This section provides a critical analysis of the contributions presented in this thesis. It exposes the main limitations and proposes solutions to overcome these constraints.

7.3.1 Java Parallel Array Interface

This section shows the main limitations of the JPAI Chapter 4.

Focused on data parallelism JPAI library is focused on data parallelism because the GPU programming model best matches this programming model. JPAI, therefore, has been designed for data parallelism in mind. However, there is no a real limitation to not include task and pipeline parallelism.

JPAI can be extended to support pipeline parallelism. JPAI would introduce a new parallel skeleton, called `pipeline`. This skeleton would receive a list of functions to be executed as different stages of the pipeline. JPAI can use composability to create the parallel pipeline of operations. One of the advantages of the pipeline skeleton is that it is possible to increase throughput between the CPU and the GPU when the computation is executed on the GPU. This can be performed through a function that copies data from the CPU to the GPU, another function for computation and a final function to copy the result from the GPU to the CPU simultaneously. The implementation of this skeleton would split the iteration space of the input data set, according to an input chunk of data that is passed as argument, and would start each stage.

Supporting task parallelism is also possible in JPAI. One possible solution is to add non-blocking methods to the API. For instance, by making the `apply` method non-blocking, the programmer can invoke a new apply for another computation and send them to the GPU concurrently. Then the result would be obtained via a blocking `get` method. This strategy would allow programmers to execute multiple OpenCL kernels in which, internally, are executed one independently of another on the GPU.

Nested parallelism Currently, JPAI does not support nested function parallelism on GPUs (e.g. two nested map parallel skeletons). The way JPAI would support it is by generating an explicit loop inside the lambda expression. Supporting a new pattern inside the map would mean to adapt the existing OpenCL code generator to obtain a new OpenCL global identifier (`get_global_id(1)`) and use the new index when the new map is encountered.

When the new parallel skeleton is invoked using the `new` Java keyword, Graal would represent it as a new object allocation node in the IR. During OpenCL code generation, the visitor would inspect the node and its type and recognise the Java `Map` object for generating a new OpenCL index.

7.3.2 OpenCL JIT Compiler and a Runtime System for Java

This section shows the main limitations of the Marawacc component presented in Chapter 5.

Parallel skeletons for OpenCL JPAI only supports the `map` parallel skeleton on the GPU. The `map` operation is one of the most important parallel skeletons that covers a wide range of applications as shown in Section 4.6. This thesis addresses the issue of how to efficiently generate OpenCL code from high-level programming languages. Other types of parallel operations, like the `reduce` operation, could be expressed as a combination of the `map` skeleton. Steuwer et. al [Steu 15] has recently shown how to solve reduction operations with a combination of multiple maps. Other parallel skeletons may require support for both, runtime and code generation, such as stencil pattern, gather or scatter. However, the compiler techniques to generate the corresponding OpenCL code for more parallel skeletons are very similar to the ones presented in this thesis.

Java Objects Marawacc does not support Java objects. However, it does recognise certain kind of Java objects, such as the Java `Tuple` object, Java arrays and object wrappers for the Java primitive types as special objects for the code generation and data management in the runtime system.

OpenCL is a parallel programming model designed for data parallel applications that operates with arrays and matrices. Arrays are the most important data type in data parallelism and GPU programming and, therefore, it is the primary data type in JPAI and Marawacc framework. Therefore, Marawacc is focused on array programming and data computing intensive applications. For more generic Java applications where any kind of object could be used, Marawacc could switch from the speculative GPU acceleration to Java threads. However, any other suitable parallel program could be used.

Supporting more Java objects, although very interesting feature, is an orthogonal problem to the runtime code generation techniques and data management optimisations.

Inheritance Inheritance is another important feature related to the object treatment that Marawacc does not support. Supporting inheritance on GPUs would need to include a virtual table (v-table) on the GPU global memory. The OpenCL kernel would track of all virtual calls. Several works have already supported this feature such as Rootbeer, JaBEE and IBM J9. However, this feature does not help to get good performance on GPUs.

Java Exceptions As GPUs do not support exceptions in hardware, Marawacc does not support Java exceptions in OpenCL. Alternately, it provides a mechanism to deoptimise (fall back the execution from the GPU to the CPU) if a runtime exception is launched. However, Marawacc provides a mechanism to recover from a GPU miss-speculation, at runtime, when a violation of the program is encountered, such as `null` pointers. This strategy is automatically performed for dynamically typed languages such as R.

7.3.3 GPU JIT Compilation for Dynamic Programming Languages

This section shows the main limitations of the OpenCL JIT compiler approach for dynamic and interpreted programming languages.

Side effects free The approach taken in this thesis to automatically offload a subset of the R program assumes that the input function to be compiled is side effects free. This assumption does not break the R semantics, because if an array is modified, R creates a new copy. However, R contains an operator (`<<-`) to update an array from an inner scope to outer scopes. The current solution presented in this thesis does not allow programmers to use this operator to guarantee that the function to compute is side effects free. The guarantee makes the parallelisation easier. However, the assumption of the input function is side effects free may break the semantic of other dynamic and interpreted languages such as Ruby.

A possible approach to overcome this limitation is with a previous data analysis before generating the OpenCL C code. This analysis should obtain the dependencies of a bigger scope than only the function to be compiled. If this analysis detects that a certain function has side effects, the runtime can set the variable to read and write in the OpenCL kernel.

Heterogeneous data types In the proposed approach of this thesis, input and output types are monomorphic. This means that the element type of an input or an output array is always the same. This is not a limitation for the R programming language, which can not mix data types for vectors. However, other such as Ruby or Python can allocate different types in a single vector. The main reason is that this is not allowed in OpenCL.

A possible solution to this limitation could be addressed by adding an indirection when accessing the OpenCL input or output elements of vectors. The OpenCL code will contain a C-struct that stores the type and the value of the corresponding type. The OpenCL kernel will receive an array of these new `structs` as a parameter instead of plain arrays. When accesses of heterogeneous types arrays are needed, the OpenCL kernel would perform an unboxing of the value, then perform the operation required and finally do the boxing into the C-struct.

7.4 Future Work

This section shows the possible future directions.

Support for 2D input data Currently, the `PArray` data structure is limited to a single dimensional array. This is not a real limitation, it is only a design decision that allows focusing on code generation and runtime performance. The main reason is that the `PArray` was designed for GPUs, where 2D or 3D matrices has to be flattened to 1D. Currently, the Java programmer has to flatten the 1D array manually.

As future work, `PArray` could be extended to process 2D and 3D `PArrays` efficiently. The JPAI could be extended to flatten arrays automatically and send them to the GPU.

OpenCL support for Java Collections None of the related approaches to automatically compile Java bytecode to GPU code, such as OpenCL or CUDA has a representation of Java collections on GPUs. One of the main limitations is that the size is unknown and GPU allocation can not be predicted. However, multiple Java applications have to be adopted and enforced to use this data structure.

JPAI and Marawacc can be extended to process fixed size of input collections and process them in batches. This is with a previous profile analysis, where the input program is executed in the bytecode interpreter. Marawacc OpenCL JIT compiler will create the equivalent Java collection on the accelerator and the runtime will reserve more space if the collection grows in size. Even if the runtime allocates more memory, the collection can also grow to overpass these limits. Therefore the runtime should provide a mechanism to detect if more memory is needed. One possible solution is that the auto-generated kernel records the actual size in another OpenCL buffer once of limit is rebased and re-allocate a new size buffer on the GPU global memory.

Multi-device support Multiple-device support opens a new world of possibilities in which execution can be mixed between multiple CPUs and GPUs at the same time to solve the same task. Marawacc can be extended to support multiple OpenCL devices to compute a single application.

To support OpenCL multi-device, the Marawacc runtime system could organise the OpenCL devices to use its own OpenCL context, OpenCL command queue and OpenCL program and kernels. In this manner, each Marawacc device (which maps the corresponding OpenCL device) can be used independently. How a runtime system (Marawacc in this case) should find the right proportion of data to execute on each device is still an open question.

Runtime OpenCL kernel specialisation One of the benefits of runtime code generation is that the architecture in which the OpenCL program will be executed is known. Marawacc, apart from the code generator, has been focused on runtime data management to improve the overall performance of Java applications. However, it has not been explored, in Marawacc, how to specialise the kernel for the underlying architecture. For example, use of different OpenCL vector types. Each hardware has different vector widths and this has an impact on performance. Another example is to explore local and constant memory with OpenCL to improve the performance of the kernel.

GPU JIT Compilation for Other Dynamic Programming Languages Chapter 6 presented a compiler technique for offloading parts of an R program into the GPU. It would be a good direction to go further in the GPU JIT compilation of other interpreted languages such as Ruby or JavaScript. As a research question, it would be good to investigate a common interface that language implementers should use if they want to offload parts of the interpreted program to the GPU. A derivative of this research is to investigate if the proposed compiler technique is performance portable across languages. This means that the performance obtained for a user application implemented in R is the same that an application implemented in another dynamic and interpreted language, such as Ruby or JavaScript, when executing on GPU.

Auto-parallelisation of Interpreted Programming Languages This thesis has focused on exploiting parallelism on GPUs for interpreted programming languages. The next step would be to investigate how to automatically identify parallelism for interpreted programming languages. Once the parallelism is detected, it can invoke the JIT compiler and the runtime system presented in this thesis to execute on GPUs.

There are many works on auto-parallelisation. Samadi et. al [Sama 14] proposed a technique for static and dynamic loop speculations to execute C applications on GPUs using CUDA. By taking advantage of the speculative compiler framework of Truffle and Graal, similar strategies can be applied to the context of dynamically typed programming languages. One possible solution would be to dynamically speculate that a part of the program is parallel and try to execute it on the GPU. If a violation is detected due a misassumption that wrongly speculated about a parallel loop, the deoptimisation process already implemented in Truffle and Graal can be used.

Appendix A

Marawacc Runtime Tables

This appendix shows the total runtime of applications presented in Section 4.6 and executed with the sizes shown in Table 4.1. It shows the total time for the Java sequential implementations, Marawacc when running on the GPU, AMD Aparapi, JOCL and OpenCL C++. The total time is reported in nanoseconds and the table shows the median runtime value of 100 executions. The purpose of reporting the median value is to make a fair comparison when running applications using the Java Virtual Machine, in which external procedures can influence when measuring performance such as garbage collection, bytecode interpreter, etc. The median value shows the total runtime of peak performance of the application when running on the JVM. This is after the initial warming-up, bytecode interpretation and compilation.

Table A1 shows the total runtime when the GPU applications are executed using the AMD R9 GPU. Table A2 shows the total runtime when the GPU applications are executed on the NVIDIA Geforce Titan Black GPU.

Size	Small	Large	Small	Large	Small	Large	Small	Large	Small	Large
	Saxpy	Saxpy	KMeans	KMeans	BS	BS	NBody	NBody	MonteCarlo	MonteCarlo
Sequential	12847575.5	90619019.5	135387494	1097721214	5726138069	23032511209	33482379968	2.15E+12	95750456433	760645819065
Marawacc	11173865	70068162	6925699	36448705	22675154	81845117	60894439.5	3294129715	184914092	1401412021
Aparapi	14043328	95638807.5	5753492	31724252	20736549.5	76369605	364725676.5	22082999971	444968494	3434413016
JOCL	10675424	69764474	6925565	38176466	19548744.5	70776657	54030078	2888655189	178579106.5	1363052534
OpenCL C++	8763100	67531100	4799790	37921600	17493400	68511300	51240300	3045600000	176104000	1360370000

Table A1: Total runtime, in nanoseconds, for execution of the Java sequential code, Marawacc, Aparapi, JOCL and OpenCL C++ on the AMD R9 GPU.

Size	Small	Large	Small	Large	Small	Large	Small	Large	Small	Large
	Saxpy	Saxpy	KMeans	KMeans	BS	BS	NBody	NBody	MonteCarlo	MonteCarlo
Sequential	12847575.5	90619019.5	135387494	1097721214	5726138069	23032511209	33482379968	2.15E+12	95750456433	760645819065
Marawacc	16344058.5	128672320	10994855	86736862	45568503.5	172747265.5	190225730.5	11765326203	263926617	2075189974
Aparapi	34234030.5	262169465	14856802.5	114661788.5	63210001.5	250989098.5	300349538	16481377673	221820368	1831870280
JOCL	16014085.5	126864822.5	11443000	89748759	32534137	128461090	57312186	3921085281	172928829	1387585641
OpenCL C++	15844700	126432000	10836200	86111300	32081200	128404000	58936600	3655650000	202585000	1319420000

Table A2: Total runtime, in nanoseconds, for execution of the Java sequential code, Marawacc, Aparapi, JOCL and OpenCL C++ on the NVIDIA GeForce Titan Black GPU.

Bibliography

[AMD 16] AMD. “Aparapi”. 2016. `http://aparapi.github.io/`.

[Baco 13] D. Bacon, R. Rabbah, and S. Shukla. “FPGA Programming for the Masses”. *Queue*, Vol. 11, No. 2, pp. 40:40–40:52, Feb. 2013.

[Bodi 09] F. Bodin and S. Bihan. “Heterogeneous Multicore Parallel Programming for Graphics Processing Units”. *Sci. Program.*, Vol. 17, No. 4, pp. 325–336, Dec. 2009.

[Brow 11] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. “A Heterogeneous Parallel Framework for Domain-Specific Languages”. In: *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, pp. 89–100, IEEE Computer Society, Washington, DC, USA, 2011.

[Carb 15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. “Apache Flink^{TM}: Stream and Batch Processing in a Single Engine”. *IEEE Data Eng. Bull.*, Vol. 38, pp. 28–38, 2015.

[Cata 11] B. Catanzaro, M. Garland, and K. Keutzer. “Copperhead: Compiling an Embedded Data Parallel Language”. In: *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, 2011.

[Chak 11] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. “Accelerating Haskell Array Codes with Multicore GPUs”. In: *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, pp. 3–14, ACM, New York, NY, USA, 2011.

[Che 09] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. “Rodinia: A Benchmark Suite for Heterogeneous Computing”. In: *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[Chib 16] S. Chiba, Y. Zhuang, and M. Scherr. “Deeply Reifying Running Code for Constructing a Domain-Specific Language”. 2016.

[Clar 17] J. Clarkson, C. Kotselidis, G. Brown, and M. Luján. *Boosting Java Performance Using GPGPUs*, pp. 59–70. Springer International Publishing, Cham, 2017.

[Clic 95] C. Click and M. Paleczny. “A Simple Graph-based Intermediate Representation”. In: *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, pp. 35–49, ACM, New York, NY, USA, 1995.

[Cole 91] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

[Coll 14] A. Collins, D. Grewe, V. Grover, S. Lee, and A. Susnea. “NOVA: A Functional Language for Data Parallelism”. In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, pp. 8:8–8:13, ACM, New York, NY, USA, 2014.

[Coop 01] K. D. Cooper, L. T. Simpson, and C. A. Vick. “Operator Strength Reduction”. *ACM Trans. Program. Lang. Syst.*, Vol. 23, No. 5, pp. 603–625, Sep. 2001.

[CUDA 07] “Nvidia CUDA”. 2007. `http://developer.nvidia.com/`.

[Cytr 91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”. *ACM Trans. Program. Lang. Syst.*, Vol. 13, No. 4, pp. 451–490, Oct. 1991.

[Dagu 98] L. Dagum and R. Menon. “OpenMP: An Industry-Standard API for Shared-Memory Programming”. *IEEE Comput. Sci. Eng.*, Vol. 5, No. 1, pp. 46–55, Jan. 1998.

[Dalo 16] B. Daloze, S. Marr, D. Bonetta, and H. Mössenböck. “Efficient and Thread-safe Objects for Dynamically-typed Languages”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 642–659, ACM, New York, NY, USA, 2016.

[Dane 15] M. Danelutto and M. Torquati. *Structured Parallel Programming with “core” FastFlow*, pp. 29–75. Springer International Publishing, Cham, 2015.

[Dean 04] J. Dean and S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pp. 10–10, USENIX Association, Berkeley, CA, USA, 2004.

[Duba 12] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. “Compiling a High-level Language for GPUs: (via Language Support for Architectures and Compilers)”. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–12, ACM, New York, NY, USA, 2012.

[Dubo 13a] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck. “Graal IR: An Extensible Declarative Intermediate Representation”. *In Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[Dubo 13b] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. “An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler”. In: *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, pp. 1–10, ACM, New York, NY, USA, 2013.

[Dubo 14] G. Duboscq, T. Würthinger, and H. Mössenböck. “Speculation Without Regret: Reducing Deoptimization Meta-data in the Graal Compiler”. In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pp. 187–193, ACM, New York, NY, USA, 2014.

[Elan 14] V. K. Elangovan, R. M. Badia, and E. Ayguadé. “Scalability and Parallel Execution of OmpSs-OpenCL Tasks on Heterogeneous CPU-GPU Environment”. In: *Proceedings of the 29th International Conference on Supercomputing - Volume 8488*, pp. 141–155, Springer-Verlag New York, Inc., New York, NY, USA, 2014.

[Enmy 10] J. Enmyren and C. W. Kessler. “SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems”. In: *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, pp. 5–14, ACM, New York, NY, USA, 2010.

[Erns 15] S. Ernsting and H. Kuchen. “Java Implementation of Data Parallel Skeletons on GPUs”. In: *PARCO*, pp. 155–164, IOS Press, 2015.

[Erns 17] S. Ernsting and H. Kuchen. “Data Parallel Algorithmic Skeletons with Accelerator Support”. *Int. J. Parallel Program.*, Vol. 45, No. 2, pp. 283–299, Apr. 2017.

[Foun 17a] H. Foundation. “HSA Platform System Architecture Specification 1.1”. 2017.

[Foun 17b] H. Foundation. “HSA Programmer’s Reference Manual: HSAIL Virtual ISA and Programming Model”. 2017.

[Foun 17c] H. Foundation. “ROCm, a new Era in Open GPU Computing”. 2017. `https://radeonopencompute.github.io/index.html`.

[Fume 13] J. J. Fumero and A. Nowak. “Vectorization with Haswell and CilkPlus”. In: *CERN OpenLab Report 2013*, 2013.

[Fume 14] J. J. Fumero, M. Steuwer, and C. Dubach. “A Composable Array Function Interface for Heterogeneous Computing in Java”. In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, pp. 44:44–44:49, ACM, New York, NY, USA, 2014.

[Fume 15] J. J. Fumero, T. Remmelg, M. Steuwer, and C. Dubach. “Runtime Code Generation and Data Management for Heterogeneous Computing in Java”. In: *Proceedings of the Principles and Practices of Programming on The Java Platform*, pp. 16–26, ACM, New York, NY, USA, 2015.

[Fume 17a] J. Fumero, M. Steuwer, L. Stadler, and C. Dubach. “Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation”. In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 60–73, ACM, New York, NY, USA, 2017.

[Fume 17b] J. Fumero, M. Steuwer, L. Stadler, and C. Dubach. “OpenCL JIT Compilation for Dynamic Programming Languages”. In: *MoreVMs Workshop. Collocated with Programing 2017*, 2017.

[Futa 99] Y. Futamura. “Partial Evaluation of Computation Process&Mdash;AnApproach to a Compiler-Compiler”. *Higher Order Symbol. Comput.*, Vol. 12, No. 4, pp. 381–391, Dec. 1999.

[Gal 06] A. Gal, C. W. Probst, and M. Franz. “HotpathVM: An Effective JIT Compiler for Resource-constrained Devices”. In: *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, pp. 144–153, ACM, New York, NY, USA, 2006.

[Gall 14] R. Gallardo, S. Hommel, S. Kannan, J. Gordon, and S. B. Zakhour. *The Java Tutorial: A Short Course on the Basics (6th Edition)*. Addison-Wesley Professional, 6th Ed., 2014.

[Geof 10] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot. “VMKit: A Substrate for Managed Runtime Environments”. In: *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 51–62, ACM, New York, NY, USA, 2010.

[Goli 13] M. Goli, J. McCall, C. Brown, V. Janjic, and K. Hammond. “Mapping parallel programs to heterogeneous CPU/GPU architectures using a Monte Carlo Tree Search”. In: *2013 IEEE Congress on Evolutionary Computation*, pp. 2932–2939, June 2013.

[Gosl 95] J. Gosling and H. McGilton. “The Java Language Environment: A White Paper”. Tech. Rep., Sun Microsystems Computer Company, Mountain View, CA, USA, Oktober 1995.

[Gril 13] L. Grillo, F. de Sande, J. J. Fumero, and R. Reyes. “Programming for GPUs: The Directive-Based Approach”. In: *Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 2013, Compiegne, France, October 28-30, 2013*, pp. 612–617, 2013.

[Grim 13] M. Grimmer, M. Rigger, L. Stadler, R. Schatz, and H. Mössenböck. “An Efficient Native Function Interface for Java”. In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pp. 35–44, ACM, New York, NY, USA, 2013.

[Grim 14] M. Grimmer, M. Rigger, R. Schatz, L. Stadler, and H. Mössenböck. “TruffleC: Dynamic Execution of C on a Java Virtual Machine”. In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pp. 17–26, ACM, New York, NY, USA, 2014.

[Han 11] T. D. Han and T. S. Abdelrahman. “hiCUDA: High-Level GPGPU Programming”. *IEEE Trans. Parallel Distrib. Syst.*, Vol. 22, No. 1, Jan. 2011.

[Henr 16] T. Henriksen, K. F. Larsen, and C. E. Oancea. “Design and GPGPU Performance of Futhark’s Redomap Construct”. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, pp. 17–24, ACM, New York, NY, USA, 2016.

[Henr 17] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea. “Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 556–571, ACM, New York, NY, USA, 2017.

[Herh 13] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. “River Trail: A Path to Parallelism in JavaScript”. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, pp. 729–744, ACM, New York, NY, USA, 2013.

[Hobe 09] J. Hoberock and N. Bell. “Thrust: A Parallel Template Library”. 2009. `http://developer.nvidia.com/thrust`.

[Holz 92] U. Hölzle, C. Chambers, and D. Ungar. “Debugging Optimized Code with Dynamic Deoptimization”. In: *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pp. 32–43, ACM, New York, NY, USA, 1992.

[Hui 90] R. K. W. Hui, K. E. Iverson, E. E. McDonnell, and A. T. Whitney. "APL\?". In: *Conference Proceedings on APL 90: For the Future*, pp. 192–200, ACM, New York, NY, USA, 1990.

[Hwu 11] W.-m. W. Hwu, Ed. *GPU Computing Gems Jade Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st Ed., 2011.

[Imam 14] S. Imam, V. Sarkar, D. Leibs, and P. B. Kessler. "Exploiting Implicit Parallelism in Dynamic Array Programming Languages". In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, pp. 1:1–1:7, ACM, New York, NY, USA, 2014.

[Ishi 15] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar. "Compiling and Optimizing Java 8 Programs for GPU Execution". In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 419–431, Oct 2015.

[JAVACL 15] "Java bindings for OpenCL". 2015. `http://javacl.googlecode.com`.

[JOCL 17] "Java bindings for OpenCL". 2017. `http://www.jocl.org/`.

[Jogamp J 17] "Jobamp JOCL". 2017. `http://jogamp.org/jocl/www/`.

[Jord 13] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer. "INSPIRE: The Insieme Parallel Intermediate Representation". In: *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, pp. 7–18, IEEE Press, Piscataway, NJ, USA, 2013.

[Kali 14] T. Kalibera, P. Maj, F. Morandat, and J. Vitek. "A Fast Abstract Syntax Tree Interpreter for R". In: *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 89–102, ACM, New York, NY, USA, 2014.

[Kall 15] M.-J. Kallen and H. Mühleisen. "Latest Developments around Renjin". Talk at R Summit & Workshop, Copenhagen, 2015.

[Kedl 14] M. N. Kedlaya, B. Robatmili, C. Caşcaval, and B. Hardekopf. "Deoptimization for Dynamic Language JITs on Typed, Stack-based Virtual Machines". In: *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 103–114, ACM, New York, NY, USA, 2014.

[Khro 17] Khronos. "SPIR-V Specification Provisional, Version 1.1". 2017. `https://www.khronos.org/registry/spir-v/specs/1.1/SPIRV.pdf`.

[Kots 17] C. Kotselidis, J. Clarkson, A. Rodchenko, A. Nisbet, J. Mawer, and M. Luján. "Heterogeneous Managed Runtime Systems: A Computer Vision Case Study". In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 74–82, ACM, New York, NY, USA, 2017.

[Kotz 08] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. "Design of the Java HotSpot&Trade; Client Compiler for Java 6". *ACM Trans. Archit. Code Optim.*, Vol. 5, No. 1, pp. 7:1–7:32, May 2008.

[Kovo 10] G. Kovoor, J. Singer, and M. Lujan. "Building a Java Map-Reduce Framework for Multi-Core Architectures". In: *In Proceedings of the Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2010.

[Krzi 16] O. Krzikalla and G. Zitzlsberger. "Code Vectorization Using Intel Array Notation". In: *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, pp. 6:1–6:8, ACM, New York, NY, USA, 2016.

[Kyse 12] V. Kysenko, K. Rupp, O. Marchenko, S. Selberherr, and A. Anisimov. "GPU-Accelerated Non-negative Matrix Factorization for Text Mining". In: *Proceedings of the 17th International Conference on Applications of Natural Language Processing and Information Systems*, pp. 158–163, Springer-Verlag, Berlin, Heidelberg, 2012.

[Lam 15] S. K. Lam, A. Pitrou, and S. Seibert. "Numba: A LLVM-based Python JIT Compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 7:1–7:6, ACM, New York, NY, USA, 2015.

[Latt 02] C. Lattner. *LLVM: An Infrastructure for Multi-Stage Optimization*. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[Lee 09] S. Lee, S.-J. Min, and R. Eigenmann. "OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization". In: *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 101–110, ACM, New York, NY, USA, 2009.

[Leop 15] D. Leopoldseder, L. Stadler, C. Wimmer, and H. Mössenböck. "Java-to-JavaScript Translation via Structured Control Flow Reconstruction of Compiler IR". In: *Proceedings of the 11th Symposium on Dynamic Languages*, pp. 91–103, ACM, New York, NY, USA, 2015.

[Lutz 13] T. Lutz, C. Fensch, and M. Cole. "PARTANS: An Autotuning Framework for Stencil Computation on multi-GPU Systems". *ACM Trans. Archit. Code Optim.*, Vol. 9, No. 4, pp. 59:1–59:24, Jan. 2013.

[Lutz 14] T. Lutz and V. Grover. "LambdaJIT: A Dynamic Compiler for Heterogeneous Optimizations of STL Algorithms". In: *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, pp. 99–108, ACM, New York, NY, USA, 2014.

[LWJGL 17] "Lightweight Java Game Library 3". 2017. `https://www.lwjgl.org/`.

[Malc 12] J. Malcolm, P. Yalamanchili, C. McClanahan, V. Venugopalakrishnan, K. Patel, and J. Melonakos. "ArrayFire: a GPU acceleration platform". 2012.

[Mani 14] S. G. Manikandan and S. Ravi. "Big Data Analysis Using Apache Hadoop". In: *2014 International Conference on IT Convergence and Security (ICITCS)*, pp. 1–4, Oct 2014.

[Marr 02] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. "Hyper-Threading Technology Architecture and Microarchitecture". Vol. 6, No. 1, pp. 4–15, Feb. 2002.

[Marr 15] S. Marr and S. Ducasse. "Tracing vs. Partial Evaluation: Comparing Meta-compilation Approaches for Self-optimizing Interpreters". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 821–839, ACM, New York, NY, USA, 2015.

[McCa 60] J. McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". *Commun. ACM*, Vol. 3, No. 4, pp. 184–195, Apr. 1960.

[McDo 13] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. "Optimising Purely Functional GPU Programs". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, pp. 49–60, ACM, New York, NY, USA, 2013.

[Mull 16] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. "Automatically Scheduling Halide Image Processing Pipelines". *ACM Trans. Graph.*, Vol. 35, No. 4, pp. 83:1–83:11, July 2016.

[NVID 09] NVIDIA. "NVIDIA Compute PTX: Parallel Thread Execution". 2009. `goo.gl/b3ybru`.

[NVID 12] NVIDIA. *CUBLAS Library User Guide*. nVidia, v5.0 Ed., Oct. 2012. `http://docs.nvidia.com/cublas/index.html`.

[Oder 04] M. Odersky and al. "An Overview of the Scala Programming Language". Tech. Rep. IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[Open 17] OpenJDK. “OpenJDK”. 2017. `http://openjdk.java.net/projects/graal/`.

[OpenCL 09] “OpenCL”. 2009. `http://www.khronos.org/opencl/`.

[Pale 01] M. Paleczny, C. Vick, and C. Click. “The Java hotspotTM Server Compiler”. In: *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, pp. 1–1, USENIX Association, Berkeley, CA, USA, 2001.

[Pita 13] U. Pitambare, A. Chauhan, and S. Malviya. “Just-in-time Acceleration of JavaScript”. In: *Technical Report, School of Informatics and Computing, Indiana University*, 2013.

[Prat 12] P. Pratt-Szeliga, J. Fawcett, and R. Welch. “Rootbeer: Seamlessly Using GPUs from Java”. In: *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pp. 375–380, June 2012.

[Raga 13] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. “Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 519–530, ACM, New York, NY, USA, 2013.

[Rang 07] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. “Evaluating MapReduce for Multi-core and Multiprocessor Systems”. In: *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 13–24, IEEE Computer Society, Washington, DC, USA, 2007.

[Ravi 13] N. Ravi, Y. Yang, T. Bao, and S. Chakradhar. “Semi-automatic Restructuring of Offloadable Tasks for Many-core Accelerators”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 12:1–12:12, ACM, New York, NY, USA, 2013.

[Rein 07] J. Reinders. *Intel Threading Building Blocks*. O’Reilly & Associates, Inc., Sebastopol, CA, USA, first Ed., 2007.

[Reye 12a] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande. “accULL: An OpenACC Implementation with CUDA and OpenCL Support”. In: *Proceedings of the 18th International Conference on Parallel Processing*, pp. 871–882, Springer-Verlag, Berlin, Heidelberg, 2012.

[Reye 12b] R. Reyes. *Directive-based Approach to Heterogeneous Computing*. PhD thesis, Universidad de La Laguna, Spain, December 2012.

[Reye 12c] R. Reyes, I. Lopez, J. J. Fumero, and F. de Sande. “accULL: An User-directed Approach to Heterogeneous Programming”. In: *Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pp. 654–661, IEEE Computer Society, Washington, DC, USA, 2012.

[Reye 13] R. Reyes, I. López, J. J. Fumero, and F. Sande. “A Preliminary Evaluation of OpenACC Implementations”. *J. Supercomput.*, Vol. 65, No. 3, pp. 1063–1075, Sep. 2013.

[Reye 15] R. Reyes and V. Lomüller. “SYCL: Single-source C++ accelerator programming”. In: *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, 1-4 September 2015, Edinburgh, Scotland, UK*, pp. 673–682, 2015.

[Rigg 16] M. Rigger, M. Grimmer, and H. Mössenböck. “Sulong - Execution of LLVM-based Languages on the JVM: Position Paper”. In: *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, pp. 7:1–7:4, ACM, New York, NY, USA, 2016.

[Robi 13] A. D. Robison. “Composable Parallel Patterns with Intel Cilk Plus”. *Computing in Science Engineering*, Vol. 15, No. 2, pp. 66–71, March 2013.

[Ross 13] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. “Dandelion: A Compiler and Runtime for Heterogeneous Systems”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 49–68, ACM, New York, NY, USA, 2013.

[Sama 14] M. Samadi, A. Hormati, J. Lee, and S. Mahlke. “Leveraging GPUs Using Cooperative Loop Speculation”. *ACM Trans. Archit. Code Optim.*, Vol. 11, No. 1, pp. 3:1–3:26, Feb. 2014.

[SDK 16] A. SDK. “Benchmarks”. 2016. `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/`.

[Seat 15] C. Seaton. *Specialising Dynamic Techniques for Implementing the Ruby Programming Language*. PhD thesis, The University of Manchester, 2015.

[Simo 15] D. Simon, C. Wimmer, B. Urban, G. Duboscq, L. Stadler, and T. Würthinger. “Snippets: Taking the High Road to a Low Level”. *ACM Trans. Archit. Code Optim.*, Vol. 12, No. 2, pp. 20:20:1–20:20:25, June 2015.

[Spri 17] M. Springer, P. Wauligmann, and M. Hidehiko. “Modular Array-Based GPU Computing in a Dynamically-Typed Language”. In: *ARRAY’17: Proceedings of the 2017 ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, 2017.

[Stad 14a] L. Stadler. *Partial Escape Analysis and Scalar Replacement for Java*. PhD thesis, Johannes Kepler University, Linz, May 2014.

[Stad 14b] L. Stadler, T. Würthinger, and H. Mössenböck. “Partial Escape Analysis and Scalar Replacement for Java”. In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 165:165–165:174, ACM, New York, NY, USA, 2014.

[Stad 16] L. Stadler, A. Welc, C. Humer, and M. Jordan. “Optimizing R Language Execution via Aggressive Speculation”. In: *Proceedings of the 12th Symposium on Dynamic Languages*, pp. 84–95, ACM, New York, NY, USA, 2016.

[Steu 11] M. Steuwer, P. Kegel, and S. Gorlatch. “SkelCL - A Portable Skeleton Library for High-Level GPU Programming”. In: *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pp. 1176–1182, IEEE Computer Society, Washington, DC, USA, 2011.

[Steu 14] M. Steuwer and S. Gorlatch. “SkelCL: A High-level Extension of OpenCL for multi-GPU Systems”. *J. Supercomput.*, Vol. 69, No. 1, pp. 25–33, July 2014.

[Steu 15] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. “Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pp. 205–217, ACM, New York, NY, USA, 2015.

[Steu 17] M. Steuwer, T. Remmelg, and C. Dubach. “Lift: A Functional Data-parallel IR for High-performance GPU Code Generation”. In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, pp. 74–85, IEEE Press, Piscataway, NJ, USA, 2017.

[Suje 14] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. “Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages”. *ACM Trans. Embed. Comput. Syst.*, Vol. 13, No. 4s, pp. 134:1–134:25, Apr. 2014.

[SYCL 16] “SYCL”. 2016. `https://www.khronos.org/opencl/sycl`.

[Talb 12] J. Talbot, Z. DeVito, and P. Hanrahan. “Riposte: A Trace-driven Compiler and Parallel VM for Vector Code in R”. In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pp. 43–52, ACM, New York, NY, USA, 2012.

[Thie 02] W. Thies, M. Karczmarek, and S. P. Amarasinghe. “StreamIt: A Language for Streaming Applications”. In: *Proceedings of the 11th International Conference on Compiler Construction*, pp. 179–196, Springer-Verlag, London, UK, UK, 2002.

[Thie 10] W. Thies and S. Amarasinghe. “An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design”. In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pp. 365–376, ACM, New York, NY, USA, 2010.

[TIOB 17] TIOBE Software. “TIOBE Programming Community Index, May 2017”. 2017. [Online; accessed 5-May-2017].

[TOP 17] TOP 500. “TOP 500, June 2017”. 2017. [Online; accessed 5-June-2017].

[Udup 09] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. “Software Pipelined Execution of Stream Programs on GPUs”. In: *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 200–209, IEEE Computer Society, Washington, DC, USA, 2009.

[Wang 14a] H. Wang, P. Wu, and D. Padua. “Optimizing R VM: Allocation Removal and Path Length Reduction via Interpreter-level Specialization”. In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 295:295–295:305, ACM, New York, NY, USA, 2014.

[Wang 14b] J. Wang, N. Rubin, and S. Yalamanchili. “ParallelJS: An Execution Framework for JavaScript on Heterogeneous Systems”. In: *Proceedings of Workshop on General Purpose Processing Using GPUs*, pp. 72:72–72:80, ACM, New York, NY, USA, 2014.

[Wang 15] H. Wang, D. Padua, and P. Wu. “Vectorization of Apply to Reduce Interpretation Overhead of R”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 400–415, ACM, New York, NY, USA, 2015.

[Wimm 13] C. Wimmer and S. Brunthaler. “ZipPy on Truffle: A Fast and Simple Implementation of Python”. In: *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, &#38; Applications: Software for Humanity*, pp. 17–18, ACM, New York, NY, USA, 2013.

[Wolf 10] M. Wolfe. “Implementing the PGI Accelerator Model”. In: *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pp. 43–50, ACM, New York, NY, USA, 2010.

[Wurt 13] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. “One VM to Rule Them All”. In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pp. 187–204, ACM, New York, NY, USA, 2013.

[Wurt 17] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. “Practical Partial Evaluation for High-performance Dynamic Language Runtimes”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 662–676, ACM, New York, NY, USA, 2017.

[Zaha 10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. “Spark: Cluster Computing with Working Sets”. In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, pp. 10–10, USENIX Association, Berkeley, CA, USA, 2010.

[Zare 12] W. Zaremba, Y. Lin, and V. Grover. “JaBEE: Framework for Object-oriented Java Bytecode Compilation and Execution on Graphics Processor Units”. In: *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pp. 74–83, ACM, New York, NY, USA, 2012.