MANCHESTER 1824
The University of Manchester

PERFORMANCE OPTIMISATIONS FOR HETEROGENEOUS MANAGED RUNTIME SYSTEMS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2021

Michail Papadimitriou

Department of Computer Science

Contents

Word Count: 35471

List of Tables

List of Figures

List of Listings

Abstract

High demand for increased computational capabilities and power efficiency has resulted in making commodity devices integrating diverse hardware resources. Desktops, laptops, and smartphones have embraced heterogeneity through multi-core Central Processing Units (CPUs), energy-efficient integrated Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), powerful discrete GPUs, and Tensor Processing Units (TPUs). To ease the programmability of these heterogeneous hardware accelerators, several parallel programming frameworks, such as OpenCL and CUDA, have been introduced, to support the new diverse computing paradigm.

In order to utilise heterogeneous hardware accelerators, software engineers shall divert from the conventional software engineering practices that until now regarded CPU-only execution. To manage this transition, a deep understanding of the underlying hardware architecture and parallel programming principles is required. To this end, several frameworks (e.g., Lift, TVM, Halide, Lime, Aparapi, Dandelion, Marawacc) made heterogeneous hardware accessible from high-level languages. Yet, these frameworks tend either to be tailored to a domain (e.g., machine learning, computer vision) or to expose hardware particularities to the developer. In addition, little work has been done on making niche aspects of heterogeneous hardware a natural extension to languages running on top of conventional managed runtimes, such as the Java Virtual Machine (JVM).

This thesis presents novel performance-oriented, architecture-dependent, compiler and runtime optimisations to enable Java applications to benefit from heterogeneous execution seamlessly. As a foundation, it uses TornadoVM, an open-source general-purpose programming framework that accommodates the execution of Java programs on heterogeneous hardware. The key objective is to bridge the performance gap between managed runtime systems and heterogeneous hardware, leading to efficient *heterogeneous managed runtimes*. This objective is fulfilled by the following three distinct contributions:

The first contribution regards the performance improvements and portability of FPGA execution. FPGAs can provide high-performance execution along with power efficiency; however, they rely on a complex process dependent on High-Level Synthesis (HLS) software. This work describes a novel approach to integrate FPGAs into high-level managed programming languages by introducing a series of runtime code specialisation techniques for seamless and execution of Java programs on FPGAs. The experimental evaluation of the FPGA execution against sequential and multithreaded Java implementations showcases a geometric mean of 1.2× with a maximum of 224× and a geometric mean of 0.14× with a maximum of 19.8× performance speedups, respectively. Furthermore, it exhibits a geometric mean for speedups of 0.32× with a maximum of 13.82× compared to TornadoVM running on an Intel integrated GPU.

The second contribution regards the automatic exploitation of the memory hierarchy of GPUs in order to increase performance. The memory hierarchy of heterogeneous hardware is a key factor for performance, yet it is complicated to exploit it, even by expert programmers. This work provides an extensible and parameterisable collection of compiler optimisations to automatically exploit data locality and the memory hierarchy for GPUs. These optimisations are implemented on top of the industrial-strength Graal compiler and enable Java programs to utilise the local memory on GPUs without explicit programming. A selection of benchmarks and GPU architectures was used to demonstrate the performance improvements. The experimental evaluation against the baseline implementations of generated parallel code, without the advantages of data locality, showcases speedups of up to 2.5×. Moreover, the new technique reached up to 97% of the performance of the native code, highlighting the efficiency of the generated code.

The third and final contribution regards the concurrent exploitation of multiple heterogeneous hardware accelerators. Heterogeneous managed runtimes need to consider application and device characteristics to perform an efficient allocation. This work addresses the seamless concurrent execution of multiple tasks on multiple devices by extending the virtualization layer of TornadoVM to execute multiple bytecode interpreters in parallel. Furthermore, the concurrent execution was combined with a machine learning model, based on a multiple-classification architecture of Extra-Trees-Classifiers, to perform efficient device-task allocation. The experimental results showcase performance improvements (up to 83%) compared to all tasks running on the best single device, while attaining up to 91% of the highest achievable performance.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgments

First of all, I would like to express my gratitude to my supervisor *Christos Kotselidis* for his guidance and support throughout the whole PhD. For the past four years, he provided me with encouragement and space to pursue my own ideas and directions. He made an already stressful experience, such as pursuing a PhD, viable while gaining valuable experiences.

Further, I would like to thank *Juan Fumero* who shared his knowledge on compilers with me, and assist me to shape my research focus. He was always willing to answer my questions and to pointed me towards the right directions. I would like to thank *Thano Stratikopoulo* for his encouragement and support during the PhD process. Also, for the effort he put to improve the quality of this thesis by patiently reviewing it.

Moreover, I would like to thank *Mary Xekalaki, Florin Blanaru, Niko Foutri and Foivo Zakkak* with who I had the opportunity to collaborate and have meaningful discussions through these years. Also, I would like to thank *James Clarkson* with whom I never worked directly, but he provided the very first version of TornadoVM.

In addition, I would like to thank *Dr. Jeremy Singer and Dr. Pavlos Petoumenos* who agreed to review my PhD and participate in my examination committee.

I would like to thank all the people I met in Manchester, especially, those from the APT group with which I had the opportunity to have a lot of interesting discussions, especially in pre-Covid times. In addition, I would like to thank the *University of Manchester, the E2Data project and HiPEAC* for providing the necessary means to support this PhD.

In addition, I would like to acknowledge all my friends back home (and abroad) that were "forcefully" stayed up-to-date with the progress of my research. A big thank is not enough to express my gratitude for their support and their comforting words all these years.

I would like to thank *my mother Alexandra, my brother Leonidas, my uncle Kostas, and my grandmother Nana* for all the love and support. I hope to be able to return even

a small portion of what they generously offered me through the years.

Also, I am grateful to *Eleni* for her support and understanding. Especially for her patience during the stressful period of putting this thesis together during Covid-19.

Last but not least, this thesis is dedicated to my grandfather *Leonidas*, who did not stay on board until the very end. I am so grateful for his unconditional love, support and understanding through the years that without it would not be possible to be here today. Among many things, not only he supported me to move abroad, but he also encouraged me to pursue this PhD from the very beginning. Thank you.

List of Abbreviations

AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
AUC	Area Under the Curve
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DL	Deep Learning
DMA	Direct Memory Access
DSP	Digital Signal Processors
FLOPS	Floating Point Operations Per Second
FPGA	Field Programmable Gate Arrays
GB	Giga Bytes
GPU	Graphics Processing Unit
I/O	Input/Output
IR	Intermediate Representation
ISA	Instruction Set Architecture
JIT	Just In Time
JVM	Java Virtual Machine
LLC	Last Level Cache
LOC	Lines of Code
LUT	Look-Up Table
MIMD	Multiple Instruction Multiple Data
ML	Machine Learning
MTMD	Multiple Tasks Multiple Devices
OOP	Object Oriented Programming
OpenCL	Open Compute Language
PCIe	Peripheral Component Interconnect Express
ROC	Reicever Operation Curve
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SSA	Single Static Assignment
SoC	System-on-Chip
TPU	Tensor Processing Unit
VM	Virtual Machine

Chapter 1

Introduction

Through the last two decades both the end of Moore's law [Moo65] and Dennard scaling [DGY+74] have resulted in novel approaches for addressing the high demand for increased computational capabilities and the need for energy efficient execution. Amongst many solutions, the integration of heterogeneous hardware accelerators has become the norm across commodity computing systems with the premise of accelerating specific workloads or parts of applications, thereby resulting in high performance and energy efficiency compared to CPUs. These computing systems with diverse characteristics span from smartphones, tablets, personal computers, and smart home devices to servers and distributed data centres. For example, nowadays, a personal computer can be equipped with a combination of multi-core Central Processing Units (CPUs), low-power integrated Graphics Processing Units (GPUs), and powerful GPUs for compute-intensive applications. In addition, large organisations seem to move away from an on-premises server infrastructures to cloud providers. This migration towards cloud-based data centres makes more niche accelerators, such as Field Programmable Gate Arrays (FPGAs) and Tensor Processing Units (TPUs), widely available. Therefore, modern developers have access to an ever increasing and diverse variety of heterogeneous hardware with an assortment of performance characteristics.

To benefit from the capabilities of this hardware, developers are required to use a mixture of parallel programming particularities along with a deep understanding of the underlying hardware architecture. Therefore, hardware expertise and knowledge of parallel programming models become the new norm, while programming practices move away from legacy CPU-oriented applications and single-core performance. For instance, Nvidia offers CUDA [NVF20], its own C-based parallel programming model to target GPU computing. In contrast, the Khronos group introduced the OpenCL

(Open Computing Language) [SGS10], a C-based framework for writing programs that execute across different heterogeneous platforms. However, most of the developers tend to prefer programming models that hide architecture-related details from them. For instance, according to the TIOBE[1] ranking system, the top three programming languages are C, Java, and Python. In addition, as in today, the top three programming languages on GitHub[2] repositories are JavaScript [Fla06], Python [VRDJ95] and Java [GJS96]. Besides, based on the `IEEE Spectrum Ranking`[3], the top three languages are Python, Java, and C with scores of 100, 95.3, and 94.6, respectively.

A potential reason for developers favouring programming languages, such as JavaScript, Python, and Java could be attributed to a number of shared characteristics between such languages. For example, they share several characteristics to provide easy accessibility by providing a high-level abstraction to interact with the hardware. These managed and interpreted languages often provide a common ground for platform portability and complete agnostic usages of low-level programming aspects, such as caching or typecasting. Therefore, managed runtime systems provide an isolated environment to handle complex aspects, such as security, class loading, garbage collection, and memory allocation seamlessly. However, these managed runtime systems focus on single or multi-core scalable performance and a high-level hardware abstraction without exploit modern heterogeneous hardware. Thus, the true benefits of heterogeneous performance remain underutilised.

Developers are willing to use heterogeneous hardware only if current programming practices do not change drastically. Moreover, conventional managed runtimes in order to adapt to this new programming environment need to evolve in a way that heterogeneous hardware becomes deeply integrated with their complete software stack. Therefore, one can strive towards `Heterogenous Managed Runtimes Systems (HMRS)` to see an intersection between conventional programming and heterogeneous hardware.

The rest of this chapter is organised as follows: Section 1.1 gives an overview of heterogeneous managed runtimes along with their adaption into several research fields. Section 1.2 outlines the key challenges of unifying heterogeneous hardware and high-level languages. Section 1.3 enumerates the three main research objectives of this thesis. Section 1.4 outlines the contributions of this thesis, while Section 1.5 presents the structure of this thesis. Finally, Section 1.6 summarises the publications that emerged from this work.

[1] `https://www.tiobe.com/tiobe-index/`
[2] `https://tinyurl.com/yzmd28vn`
[3] `https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020`

1.1 The Advent of Heterogeneous Managed Runtimes

Heterogenous managed runtimes [CFP+18a] are an emerging paradigm that bridges the gap between conventional virtual machines, such as the JVM and heterogeneous hardware, such as GPUs and FPGAs. These new heterogeneous managed runtimes can be successfully used across a broad range of fields, as many applications written in high-level languages can benefit from heterogeneous hardware. There are three pillars of moving away from the conventional managed runtimes: a) the large amounts of data that needs to be processed, b) the availability of hardware with diverse characteristics, and c) the need for real-time processing. However, one needs to guarantee not only seamless execution, like in conventional homogeneous systems, but also high utilisation of the underlying hardware.

Researchers and industry practitioners innovated on systems capable of mitigating the challenges of the transition to high-level abstractions of heterogeneous hardware for domain-specific acceleration. Apache TVM [CMJ+18] is a solution that enables heterogeneous execution for deep learning frameworks, such as Keras [C+15], MXNet [CLL+15], PyTorch [PGM+19], Tensorflow [AAB+15], CoreML [Tha19] and DarkNet [Red16] with a single-entry point for compilation and execution to different backends, including browsers, microcontrollers, FPGAs and GPUs. In addition, it provides support for all the popular front-ends of the deep learning frameworks, such as Python, C++, Rust, Go, Java, and JavaScript. In same direction, another domain with a high demand for computational power, the image processing domain benefits from the Halide [RKBA+13] programming language. Halide targets high-performance image and array processing through execution with CUDA [NVF20], OpenCL [SGS10], OpenGL Compute Shaders [WNDS99] and Apple Metal [App]. Recently, the Big Data analytics domain try to mitigate the challenges and strive towards exploiting heterogeneous hardware [XFK18, KDA+20]. In more detail, the Apache Spark [ZXW+16] merit from heterogeneous hardware and execution with the ShadowVM [LHWW21]. ShadowVM allows the JVM to be augmented with a minimal runtime to support concurrent CPU/GPU execution for Resilient Distributed Datasets (RDDs) in Spark.

On the contrary, several frameworks provide high level languages with accessibility to general purpose acceleration through heterogeneous hardware. To achieve this seamless execution and parallelisation they employ multiple different approaches to exploit parallelism. These approaches vary from exposing low level parallel computing primitives to the language semantics to loop parallelism for `for` loops, `while` loops, and lambda expressions with the parallel `streams`. For instance, Sumatra [Orab],

IBM J9 [IHKS15], Aparapi [SCN+15], Lime [DCR+12, ABB+12], Jacc [CKBL15], Marawacc [FRSD15, Fum17] and TornadoVM [CFP+18b, FPZ+19] target heterogeneous execution for applications written in Java. In addition, other frameworks, such as Hot&Spicy [SMSF18] and ALPyNA [JS19, JTS19a] allow Python programs to be executed on GPUs and/or FPGAs. Furthermore, for C#, frameworks, such as Hybridizer [Alt] provide support for advanced features, such as `Parallel.For`, generics or virtual functions while targeting only Nvidia GPUs. On the downside, these frameworks tend to narrow down the scope either of the language support or of the device support. For example, they can either target execution on GPUs or FPGAs due to the differences in terms of programmability and software requirements (i.e., High Level Synthesis (HLS)).

Domain-specific heterogeneous programming frameworks, such as TVM [4] and Halide [5] gain a lot of traction. However, general-purpose approaches, such as TornadoVM, remain primarily an academic pursuit. The key challenge of making heterogeneous managed runtimes accessible for general-purpose acceleration is the need of making hardware as transparent as possible through compiler and runtime optimisations. The following section summarises three key outstanding problems in making high-performing hardware acceleration accessible from high level languages through heterogeneous managed runtimes.

1.2 Challenges in Heterogeneous Managed Runtimes

As previously stated, the rapid advance of heterogeneous hardware unveiled an extensive gap between widely adopted programming languages and heterogeneous parallel programming constructs. Heterogeneous managed runtimes attempt to mitigate this transition towards heterogeneous hardware. However, this process imposes the following challenges:

Runtime Support for Niche Accelerators

The advent of heterogeneous computing led to a plethora of hardware devices. For instance, CPUs, GPUs, FPGAs, TPUs and ASICs are all available as standalone compute units or combined. CPUs and GPUs have a predefined ISA, TPUs and ASICs a predefined functionality for domain-specific purposes, and FPGAs lie in the middle.

[4] ≈ 6.700 stars in Github: `https://github.com/apache/tvm`
[5] ≈ 4.300 stars in Github: `https://github.com/halide/Halide`

FPGAs can provide the high-performance of ASICs along with a multiple programming choices similar to GPUs. However, they are typically requiring deep understanding of the HLS software along with the particularities of each of the hardware vendors.

In order for FPGAs to be accessible from managed runtimes, there is a need mitigate the programmability overhead of the HLS software. In prior work [CDL13, KPZ+16, KFP+18, MCC18, RKBA+13, BRR+19, SG08, SMSF18, DBAS18, RYC+13], systems, in order to enable FPGA execution for high-level languages either provide support for specific operations through pre-compilied binaries or through domain-specific languages (DSLs). Therefore, the lack of support for dynamic compilation and an easy way to integrate with managed runtimes, prevents FPGAs to be used for general-purpose acceleration.

Compiler Support for High Performance Code Generation

Being able to generate efficient and high-performance heterogeneous code from high-level languages imposes significant challenges. For instance, conventional CPU architectures rely on managed runtimes, such as the Common Language Runtime (CLR) and the JVM for automatic memory management. However, for hardware architectures, such as GPUs where memory allocation is explicit, exploiting the memory hierarchy is crucial. Hence, this process becomes more complex when one tries to target GPUs from high-level languages in which memory allocation is being managed by the runtime.

A trade-off exists between programmability, compilation time, and the attainable performance. For example, iterative compilation processes can generate high-quality code, but with prolonged combination times. On the contrary, the exposure of hardware particularities to the developer can mitigate the prolonged compilation times, while sacrificing programmability. As a result, there is a compromise between programmability and performance. Hence, there is a need to provide compiler support for high performance code generation from high-level languages.

Beyond Single Device Execution

Given the large variety of hardware devices, many researchers invested in making a single heterogeneous device accessible from a high-level language. However, little research has been done in the area of automatic utilisation of more than one heterogeneous device concurrently. Since generating high-performance parallel code is a challenging task for compilers, handling the execution and compilation for multiple

devices also requires runtime support.

Novel software techniques must be developed to allow heterogeneous managed runtimes to run applications on multiple devices concurrently. However, regardless the invested effort on the compilers of these systems, not all applications are mapped efficiently to all devices due to the diverse computational requirements and architectures. Therefore, performing the device allocation intelligently is crucial to achieve high performance through multi-device execution.

1.3 Research Objectives

The general aim of this thesis is to conduct research on novel techniques and optimisations to enable existing managed runtimes to achieve high performance, while executing on heterogeneous hardware. As the research space spans into a broad multidisciplinary area (compilers, runtime, programming languages, hardware), the general aim has been narrowed down to three sub-aims covering different areas, and addressing the challenges highlighted in Section 1.2. In detail, the research objectives of this thesis are:

1. Explore the opportunities for increased performance while leveraging FPGAs through a managed language, such as Java, without sacrificing programmability. This work mitigates the complexity associated with FPGA execution due to High-Level Synthesis (HLS) software that requires understanding specifics of the underlying hardware. The practical goal of this research is to allow Java programs to be executed on FPGAs without requiring from end-users to program FPGA specific code. This work provides a natural integration with the Java ecosystem and provide an easy approach for deployment.

2. Investigate how a combination of optimisations during Just-in-Time (JIT) compilation and at run-time can lead to high-performing GPU-targeted code generation by exploiting the memory hierarchy GPUs. The main target is a managed runtime language, such as Java, without sacrificing programmability. These optimisation techniques allow usability and integration with industrial quality compilers, such as Graal [DWS+13]. The practical goal of this work is to provide an infrastructure yet sufficient to cater multiple GPU-oriented optimisations.

3. Provide a single heterogeneous managed runtime environment capable of scheduling tasks on multiple heterogeneous devices concurrently. Having a system

capable to perform such execution allow the exploitation of different scheduling approaches and modelling techniques based on precise device characteristics. The practical goal of this research is to provide enough and detailed information on the merits of heterogeneous concurrency while documenting the potential performance gains against single device execution and different multiple device allocations.

1.4 Contributions

This thesis presents several novel approaches to bridge the performance gap between current managed runtime systems (in particular the JVM) and heterogeneous hardware accelerators. The main goal is to leverage existing techniques and introduce novel ones to showcase the performance benefits of applications written in a managed programming language, such as Java, from heterogeneous hardware. Hence, this thesis aims to make heterogeneous hardware widely accessible without relying on a deep understanding of the underlying hardware architecture. This thesis makes the following contributions:

- It introduces an open-source end-to-end toolchain designed to transparently compile and run Java code on FPGAs through the introduction of a set of execution modes from the Java Virtual Machine (JVM). This approach enables developers without FPGA expertise to harness the execution capabilities of FPGAs. This toolchain also provides seamless integration with emulation tools provided by the HLS vendors to enable fast prototyping. It combines a compiler extension that automatically exploits parallelism from sequential code at the IR level and specialises the IR for OpenCL code generation targeting FPGAs. This is achieved by introducing compiler phases able to augment and transform the IR with FPGA-specific pragmas. Finally, it showcases end-to-end speedups of up to 19.8× and 224× over multi-threaded and sequential Java code, respectively.

- It presents a new optimisation scheme integrated with a widely adopted industrial JIT compiler to perform automatic local memory allocation on GPUs. This capability is enabled by introducing the notion of *compositional compiler intrinsics* on top of Java snippets to perform parametrised data locality optimisations targeting GPU code generation. Results showcase performance improvements of up to 2.5× versus the original code produced by TornadoVM; while also reaching up

to 97% of the performance of the manually optimised code across three different GPU vendors.

- It presents a novel mechanism to enable Multiple-Tasks Multiple-Devices (MTMD) execution for Java programs by utilising the available OpenCL-compatible devices in a system, while running a standard JVM.

- It presents a new open-source static feature extraction tool capable of obtaining specific performance metrics from standardised Graal IR.

- It presents a machine learning system based on a multiple classifier model capable of performing high-performing task allocation onto a device selected among CPUs, integrated GPUs, and discrete GPUs. Moreover, it presents an experimental evaluation of this approach, showcasing the benefits of concurrent execution and prediction-based allocation. Finally, the system achieved up to 83% performance improvements against the best single device and up to 91% of the best concurrent configuration.

1.5 Thesis Structure

The remainder of the thesis is organised as follows:

- **Chapter 2** provides background information on the main concepts, techniques, technologies, and platforms used in this thesis. Besides, it explains the terminology required to discuss the current state-of-the-art research in the scope of this thesis.

- **Chapter 3** reviews the state-of-the-art work relevant to the contributions of this thesis. It analyses approaches that focus on making FPGA execution feasible for high-level managed languages. Also, it analyses compilation techniques that aim to improve data locality on GPUs. Finally, it discusses several approaches that investigate multi-device execution of parallel workloads.

- **Chapter 4** describes how a heterogeneous managed runtime system is augmented to support FPGA execution for Java programs. It details how the proposed platform integrates several techniques, such as two-stage JIT compilation and compiler optimisations.

- **Chapter 5** presents a novel combination of JIT compilation optimisation techniques to exploit data-locality for GPU code generation efficiently. It describes how various levels of the GPU memory hierarchy can be utilised seamlessly through the introduced optimisations. Finally, it evaluates the proposed optimisations against hand-written optimised code.

- **Chapter 6** presents a novel mechanism to enable the execution of multiple tasks on multiple devices for Java applications, concurrently. It illustrates in detail the performance shortcomings of statically allocating tasks to devices without exploiting intelligent allocation strategies. Also, it outlines the implementation of a static feature extractor tool from a Java compiler graph and machine learning architecture to perform the near-optimal device-task allocation. Finally, it presents an evaluation of the complete system against several scheduling approaches.

- **Chapter 7** summarises the contributions of this thesis and discusses opportunities for future work and research directions.

1.6 Publications

Parts of the work in **Chapter 4** were used in the following publications:

1 **Michail Papadimitriou**, Juan Fumero, Athanasios Stratikopoulos, Christos Kotselidis. *Transparent Compiler and Runtime Specialisations for Accelerating Managed Languages on FPGAs*. In The Art, Science, and Engineering of Programming, 2021, Vol. 5, Issue 2, Article 8, 2020 (Programming'21) [PFS+20].

2 **Michail Papadimitriou**, Juan Fumero, Athanasios Stratikopoulos, Christos Kotselidis. *Enabling Prototyping and Acceleration of Java Programs onto Intel FPGAs*. In Proceedings of the IEEE International Symposium On Field-Programmable Custom Computing Machines, 2019. (FCCM'19) [PFSK19].

3 **Michail Papadimitriou**, Juan Fumero, Christos Kotselidis. *Exploiting Programmability of FPGAs Through Managed Runtime Systems*. In Proceedings of the International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems, (ACACES'18) [PFK18].

4 Juan Fumero, **Michail Papadimitriou**, Foivos Zakkak, Maria Xekalaki, James

Clarkson, Christos Kotselidis.*Dynamic Application Reconfiguration on Heterogeneous Hardware*. In Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. 2019. (VEE'19) [FPZ+19].

Also, part of the work presented in **Chapter 5** was used in the following publication:

5 **Michail Papadimitriou**, Juan Fumero, Athanasios Stratikopoulos, Christos Kotselidis. *Automatically Exploiting the Memory Hierarchy of GPUs through Just-in-Time Compilation*. In Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. 2021. (VEE'21) [PFSK21].

Part of the work used in **Chapter 6** was used in the following publication:

6 **Michail Papadimitriou**, Eleni Markou, Juan Fumero, Athanasios Stratikopoulos, Florin Blanaru, Christos Kotselidis. *Multiple-Tasks on Multiple-Devices (MTMD): Exploiting Concurrency in Heterogeneous Managed Runtimes*. In Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. 2021. (VEE'21) [PMF+21].

In addition, parts of this work have been published to the following research papers:

7 James Clarkson, Juan Fumero, **Michail Papadimitriou**, Foivos S Zakkak, Maria Xekalaki, Christos Kotselidis, Mikel Luján. *Exploiting High-Performance Heterogeneous Hardware for Java Programs using Graal*. In Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Managed Languages and Runtimes (ManLang). 2019. (ManLang'19) [CFP+18b].

8 James Clarkson, Juan Fumero, **Michail Papadimitriou**, Foivos S Zakkak, Maria Xekalaki, Christos Kotselidis, Mikel Luján. *Towards practical heterogeneous virtual machines*. MoreVMs 2018, (MoreVMs'18) [CFP+18a].

Moreover, the work of this thesis has been put forward into two research competitions:

- **Michail Papadimitriou**. *Towards Multi-Device Concurrent Heterogeneous Execution*. MICRO 2020 ACM Student Research[6] Competition.
- **Michail Papadimitriou**. *Exploiting Reconfigurable Hardware from Managed Languages*. **(One of the winning submissions)** ACCELCLOUD: International Contest on Accelerated Heterogeneous Cloud Computing, HiPEAC, 2019.

[6] `https://www.microarch.org/micro53/program/src.php`

1.7 Summary

This chapter provided a brief introduction to the research domain of this thesis. Also, it outlined the challenges, research aims, contributions and publications that appearing in this work. The next two chapters discuss the background knowledge required to comprehend the content of this thesis and the state-of-the-art work. The subsequent chapters describe novel techniques which address the three challenges of runtime support for niche accelerators, compiler support for high performance code generation and beyond single device execution.

Chapter 2

Background

This chapter provides the required background knowledge for the reader to comprehend the material discussed in the following chapters of this thesis. Section 2.1 gives an overview of the available heterogeneous platforms in modern computer systems, along with the respective programming models. In more detail, Section 2.1.1 provides information about the hardware aspects of CPUs, GPUs, and FPGAs, while Section 2.1.2 outlines the main heterogeneous programming aspects. As this work highly depends on Managed Runtimes and Java, Section 2.2 explains their primary software components. Finally, Section 2.3 outlines the new trend of Heterogeneous Virtual Machines, and in more detail, the concept of TornadoVM, the experimental research platform used in this thesis.

2.1 Heterogeneous Platforms

Dennard's scaling [Boh07] and the abrupt end of Moore's Law [TW17] led to the combination of parallel architectures with different performance and power characteristics to be a necessity in order to attain higher performance and energy efficiency. The advent of a new computing paradigm has emerged through this new era, known as **heterogeneous** computing. In this new computing paradigm, system design and implementation move away from scaling to multiple unified cores to combine several specialised ones for specific tasks. For instance, hardware vendors, such as Intel, ARM, Nvidia and Xilinx have recently shipped powerful CPUs with low-power integrated GPUs and FPGAs on the same die [Int21]. These platforms can yield high performance and power efficiency as all devices can operate simultaneously and accelerate various compute-intensive applications or different parts of the same applications. However, these heterogeneous

platforms require an application to be written through heterogeneous programming languages to achieve performance improvements.

Programmers carry the burden of explicitly exploiting the available parallelism to harness the potential capabilities of the parallel hardware. The existence of this plethora of diverse architectures with a variety of characteristics in terms of performance or power consumption led to several parallel programming models. Each of the different architectures has several parallel programming models available. For instance, multicore CPUs can be programmed with OpenMP, while Nvidia GPUs can be targeted with CUDA. Also, more niche accelerators, like FPGAs require programs to be written with low-level hardware description languages, such as Verilog [STDP19].

2.1.1 Heterogeneous Architectures

2.1.1.1 Multi-Core Central Processing Units (CPUs)

Modern CPU architectures have adopted multicore designs. A single die contains up to several physical cores capable of performing independent computations simultaneously. These physical cores embed several processing units tightly or loosely with shared-memory inter-core communication protocols, such as the Intel QuickPath Interconnect [ZBMS10].

Multicore CPUs designs have been augmented with unique Single Instruction Multiple Data (SIMD) instruction sets, such as AVX [Lom11], or SSE [Intd] to increase the Instruction Level Parallelism (ILP). Besides, each core contains several levels of caches for data and instructions, along with vector floating-point units (VPUs) and Fused Multiply-Add (FMA) operations to optimise performance. Therefore, modern CPUs have been widely adopted by different computing domains for modern computing due to the extensive availability of tools, programming languages, and educational resources.

However, lately, even the multicore CPU architectures have been shifting towards heterogeneity to create a multicore processor that can adapt quickly between high performance and low power profiles. A prime example is the ARM big.LITTLE architecture [ARM21], featuring "big" cores for high-performance and "LITTLE" cores for low power consumption.

2.1.1.2 Graphics Processing Units (GPUs)

Graphics Processor Units (GPUs) are devices with a unique architecture capable of achieving high computational throughput, while utilising their highly parallel structures. The modern GPU architectures contain thousands of homogenous cores capable of deploying thousands of threads. The Nvidia GPU architecture is one of the most dominant [1] in the current computing era. It targets a broad range of applications varying from personal computers for gaming and rendering up to server solutions for specialised computations, such as machine learning modelling. However, other vendors like AMD, and lately, Intel with the Intel Iris Xe [Inta] discrete GPU, offer hardware with different compute capabilities. Regarding the NVIDIA GPU model, it leverages the Single Instruction Multiple Threads (SIMT) execution model. This architectural model uses Streaming Multiprocessors (SMs), each of them equipped with multiple threads. In the NVIDIA context, these multiple threads are grouped in packs of 32, also known as *warps*. *Warps* implement the SIMT model, so the same instruction is executed by multiple threads on multiple data.

GPUs also provide their memory hierarchy model compared to normal CPU architectures. GPUs are usually equipped with three different levels of memory of diverse sizes and access speeds. Off-chip memory is the one that provides available capacities up to several Gigabytes (GBs) of DRAM accessible by all threads. However, performance can be significantly impacted when all threads require simultaneous access to the DRAM. To improve performance through data locality, GPUs also employ a smaller size fast cache shared among the same SM threads. However, this needs to exploit by the software developer explicitly (unlike their implicit use in CPUs). Finally, the fastest memory is available in the register files. Each thread has its register file that can store temporary data used by every single thread.

Driven by the need for low-power devices, integrated low-power GPUs have been adopted in several computing systems; albeit with, lower performance compared to the discrete GPUs. Integrated GPUs tend to be fabricated on the same die with CPUs and share the main memory (RAM) of the system. This design increases the heterogeneity factor available in widely available computing devices. Some noticeable integrated GPU micro-architectures are the Intel HD graphics series [Intb], the ARM Mali GPUs [Ltd].

[1] As in today: `https://www.t4.ai/industry/gpu-market-share`

2.1.1.3 Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays (FPGA) are special-purpose reconfigurable circuits that the end-user can program for specific operations under demand. FPGAs can be reconfigured during run-time in contrast to more specialised architectures, like ASICs. In addition, FPGAs provide higher energy efficiency than general-purpose cores but lower than ASICs for specific workloads [BTL10]. On-device resources are limited, and one needs to achieve high hardware utilisation to achieve high performance. However, this is a challenging process as it requires advanced knowledge of underlying circuitry and the use of hardware description languages, like VHDL [Per93] and Verilog [TM96]. Also, the compilation (i.e., bitstream generation) process is a bottleneck, as the creation of the executables can take up to hours.

FPGAs consist of a large number of configurable logic blocks (CLBs) and look-up tables (LUTs). Each CLB contains a set of LUTs, a configurable switch matrix, a selection circuit (MUX), registers, and flip-flops. All these resources are connected via programmable interconnects. LUTs are large hardcoded truth tables and often are used as small memory components through high-speed indexing. SRAM blocks are interspersed in the fabric and can be chained together to build deeper, wider memories or RAMs. FPGA vendors offer suites with highly optimised circuitry components (i.e., IPs), such as DSPs and filters, to speed up the design process. Modern FPGAs target low latency (e.g., high-frequency trading) applications using multi-gigabit transceivers (MGTs) [STDP19].

Moreover, following the trend of heterogeneity, FPGAs lately integrated into various setups, such as embedded systems and servers. In the embedded systems market, vendors have shipped FPGAs in the same chip combined with low-power CPUs, known as System-on-Chip (SoC). Also, FPGAs have been added into servers via the high-speed PCIe Interconnect for high-performance computing. For instance, examples such heterogeneous platform are the Xilinx Zynq-7000 [RBD+11] and the Xeon CPU-Arria 10 FPGA [Int20] hybrid chip from Intel.

2.1.2 Parallel Programming Models

To be able to utilise efficiently the plethora of the heterogeneous architectures, as well as their unique performance features, developers employ parallel programming models. These parallel programming models allow developers to write parallel software with exposure to low-level architectural primitives. Currently, various parallel programming

frameworks exist, offering different levels of exposure to the underlying architecture. For instance, at a *binary-level* with the SPIR-V [Gro] which is a modern cross-platform low-level intermediate language and at *source-level* with the OpenCL programming language. In addition, approaches at the *source-level* often complement widely adopted programming languages, such as C++ while offering an abstraction from the low-level primitives. For example, Nvidia GPUs can be programmed with CUDA [NVF20], while the OpenCL model can target a variety of devices, like CPUs, GPUs, and FPGAs. Finally, the OpenMP [CDK+01] model that allows scaling to thousands of uniform cores through multi-platform shared-memory parallel programming.

This thesis investigates how managed runtimes can use various heterogeneous architectures and it uses the OpenCL parallel programming model as it provides platform portability. Therefore, the following section briefly exploits its characteristics.

2.1.2.1 OpenCL

The Open Computing Language (OpenCL) [SGS10] is an open standard managed by the Khronos [2] group. Briefly, OpenCL provides a system for expressing parallelism and portability of computation workloads across several heterogeneous devices, such as GPUs, CPUs and FPGAs. At the conceptual level, the OpenCL architecture is divided into the following *four* distinct models [RBD+11]:

- **Platform Model:** How the compute devices are internally organised, and how they connect to a host.
- **Execution Model:** How the workloads are being executed on the device.
- **Memory Model:** How the data is stored and organised between the device and the host, and it provides a standardised convention for the memory hierarchy.
- **Programming Model:** The data-parallel and task-parallel programming approaches that the underlying execution model supports.

Platform Model: The OpenCL platform model provides a host that is able to connect to one or more OpenCL-compatible devices. The same model provides an abstract definition for how these devices are organised internally. Figure 2.1 depicts a rough overview of the platform model. The host is connected with several *Compute Devices*. Each device consists of many *Compute Units (CU)* that are mapped to the

[2] https://www.khronos.org/

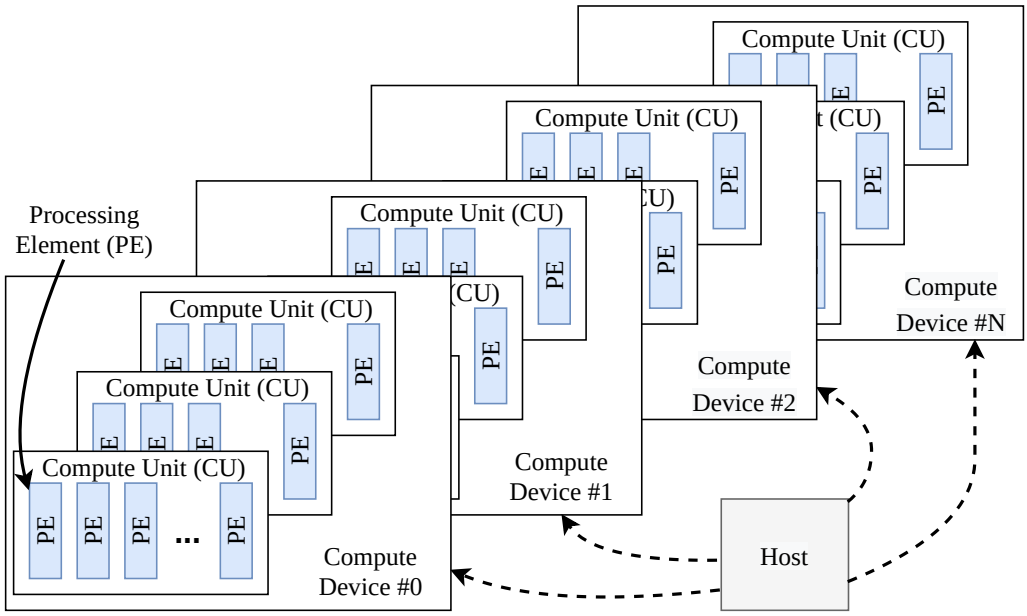


Figure 2.1: Overview of the OpenCL host-device relations.

physical cores of the device. Additionally, each of these *Compute Units* can contain several *Processing Elements (PEs)* mapped into threads.

Execution Model: The OpenCL execution model consists of two parts, the *host program* and the *kernel*. The host programs are executed by the main system, while at least one OpenCL-compatible device execute the kernel. The host program is used to orchestrate the data transfers to/from the device along with the kernel configuration and execution. In OpenCL, the kernel is a C-based function stored in a separate source file (i.e., with `.cl` extension), and it expresses a segment of parallel code that targets a heterogeneous device. However, a key difference with standard C code is that OpenCL introduces several parallel intrinsics to organise operations that are performed in parallel.

Figure 2.2 illustrates how the available compute items are organised through the execution model. Each device has a limit on the available threads, the so-called *work-items*, configured into a three-dimensional layout. The index space of these *work-items* reflects to the *NDRange*. *NDRange* is defined as an N-dimensional integer array with each of its entries corresponding to a *GlobalID* of each distinctive *work-item*. These *work-items* are grouped into *work-groups*, with each of the *work-groups* preserving a unique ID. Also, *work-items* that live in a specific *work-group* have also a local ID.

Additionally, on the host side, the application needs to define a context of orchestrating the execution of the kernels. The OpenCL execution context focuses on four distinct components:

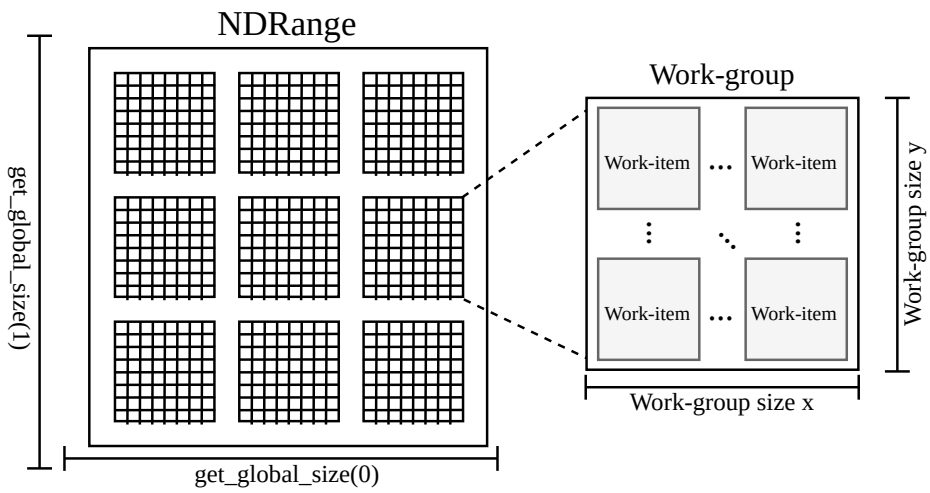


Figure 2.2: Overview of the OpenCL execution model. NDRange index space showing work-items, their global IDs and their mapping onto the pair of work-group and local IDs.

- **Devices:** The OpenCL-compatible devices to be available from the host. Figure 2.1 showcases a host system with access to multiple OpenCL-compatible devices from different hardware vendors.
- **Kernels:** A selection of C-based function written explicitly with the OpenCL API.
- **Program Objects:** The source code or the machine code in a binary format that implement the functions mentioned above. OpenCL kernels rely on program objects as the source code is compiled through vendor-specific compilers.
- **Memory Objects:** These objects store the values and host pointers of the data needed to perform the computation encapsulated in the kernel.

The host preserves a *command-queue* of the kernels waiting to be executed. These *command-queues* are responsible for the kernel execution, memory, and synchronisation commands. Moreover, these commands can be invoked in an *in-order* or an *out-of-order* sequence.

Memory Model: The OpenCL memory hierarchy is similar to the memory hierarchy of conventional CPU architectures. Figure 2.3 displays the OpenCL memory hierarchy that is organised in four distinct levels; 1) *Global Memory*, 2) *Constant Memory*, 3) *Local Memory*, and 4) *Private Memory*. *Global Memory* provides a space that allows read/write access to all work-items running in parallel. Also, the *Global*

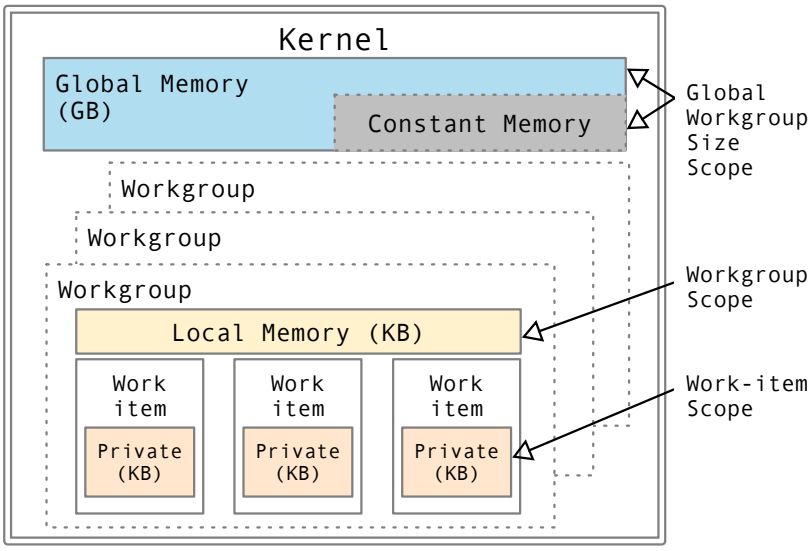


Figure 2.3: Overview of the OpenCL memory model. Consistency within work–group for global and local memory: Only at synchronization points within work–group and Consistency between work–groups for global memory: Only at synchronization points at the host level.

Memory has a special region called *Constant Memory*, which can be allocated for read-only data. The next memory tier is the *Local Memory*, which can be accessed (read/write) by all work-items in the same work-group with explicit synchronisation through barriers [FSV14]. Finally, the last memory tier is the *Private Memory* which belongs exclusively to a single work-item for storing data to a number of registers.

Table 2.1 showcases the access status of each memory region. The *Global Memory* (in the range of GBs) corresponds to the main memory, while *Lobal Memory* (up to hundreds of KBs) corresponds to the L2 cache. Finally, *Private Memory* (up to tens of KBs) is exclusive to each work-item, and it is therefore equivalent to the L1 cache of a standard CPU. However, unlike conventional CPUs, which have hardware cache coherency support, the OpenCL memory model requires communication barriers for

Table 2.1: Overview of the allocation and access support for the different OpenCL memory regions from the device and host perspectives.

Side	*Specs*	*Memory Region*			
		Global	**Constant**	**Local**	**Private**
Host	Allocation	Dynamic	Dynamic	Dynamic	Not-accesible
	Access	Read/Write	Read/Write	No access	No access
Kernel	Allocation	Not-accesible	Static	Static	Static
	Access	Read/Write	Read-only	Read/Write	Read/Write

coherency. In addition, the access latency to each region varies in the range from ~40 to ~450 cycles for local and global memory, respectively [WPSM10].

Programming Model: The open standard allows the user to have access to both data-parallel and task-parallel programming models. However, the primary focus of OpenCL is on data parallelism.

2.2 Managed Runtime Systems

This thesis focuses on using the Java programming language, the Java Virtual Machine (JVM), and its optimising compiler. This section explains the main concepts related to the JVM and how it works. Also, it presents Graal, an aggressive JIT compiler framework for the JVM, along with a discussion of specific optimisation utilities, such as the compiler Snippets [SWU+15].

2.2.1 Java

Java [AGH05] is a general-purpose, object-oriented programming language developed by Sun Microsystems, released in 1995. Through the years, Java has gained a lot of traction and has become dominant among other programming languages, and according to the TIOBE ranking system, it was ranked 2nd by the July of 2021[3]. Today, a large number of devices, such as personal computers, gaming consoles, smartphones, and Internet applications, use Java.

2.2.2 Java Virtual Machine (JVM)

Originally, the main goal of Java was to introduce a "write once, run anywhere" approach to allow programs to execute onto a broad range of hardware and operating systems. To achieve this functionality, the Java Virtual Machine (JVM) platform emerged. The JVM guarantees the interoperability of Java programs across different operating systems and computer architectures.

Briefly, developers provide their applications written in a structured way with Java. This source code is written in an architecture-independent manner. The Java Compiler `(javac)` compiles the Java source code into `class` files that hold the corresponding Java bytecodes [Oraa]. In addition, multiple class files can be combined and packaged together in a Java Archive (a so-called jar-file). These bytecodes, once are loaded

[3] `https://www.tiobe.com/tiobe-index/`

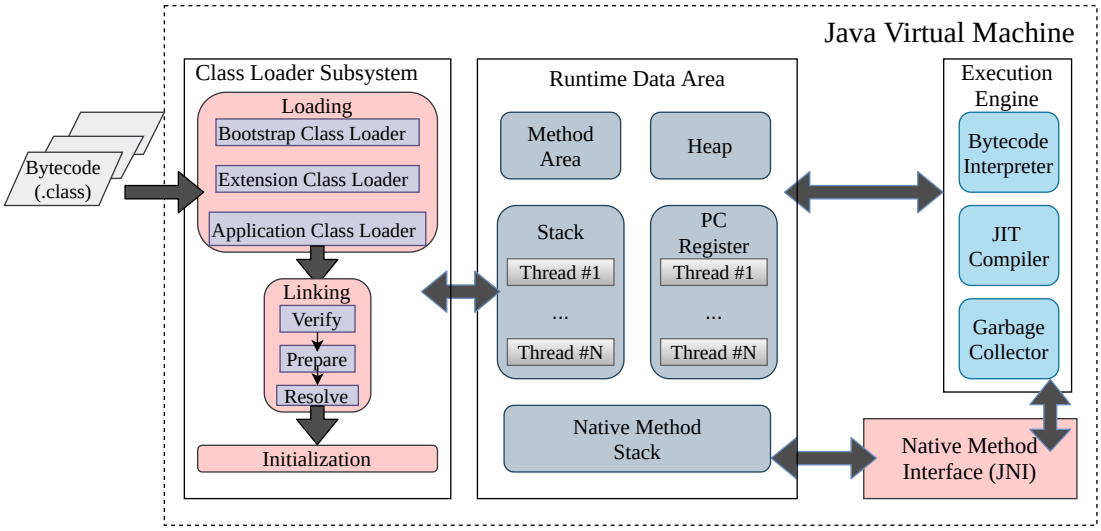


Figure 2.4: Java Virtual Machine (JVM) architecture.

on the JVM, the interpretation starts. During interpretation, the interpreter executes the bytecodes in deterministic order without applying any optimisations. However, during the execution, bytecodes corresponding to methods that are called repeatedly (i.e., known as a *hot methods*) can be Just in Time (JIT) compiled.

Figure 2.4 depicts an abstract overview for the architecture of the Java Virtual Machine. This architecture consists of three main components: 1) Class Loader, 2) Runtime, and 3) Execution Engine.

Class Loader: The class loading process has three different types of loaders:

1. **Bootstrap Class Loader:** Loads core and Java Development Kit (JDK) classes.

2. **Extension Class Loader:** Loads classes from the JDK extensions directory.

3. **System Class Loader:** Loads classes from the system class path.

Then there is a Linking process for classes or interfaces. When *class loading* is finished, there is a verification and preparation process for these classes or interfaces. If needed, the direct superclasses and superinterfaces are also being prepared. Linking ensures that each class or interface is completely loaded, verified, and prepared before initialisation occurs.

The rest of the components can be described briefly by the following:

- **The Runtime Data Area** that defines various run-time data areas that are used during the execution of a program. These are the class area, the Java Heap, the

stack and PC registers. All the above store information regarding object states, as well as the state of the JVM during execution.

- **The Execution Engine** that executes the bytecodes produced by the Runtime Data Area. Three main components orchestrate the bytecode execution at this stage. The Interpreter, the JIT compiler, and the Garbage Collector.
- **The Interpreter** that interprets the bytecodes fast, but the execution is slow.
- **The JIT compiler** that complements the interpreter. For methods calls that occur repeatedly, the JIT compiler is going to identify and specialise these calls. Then native code is going to be produced and used directly to lower the cost of continuous interpretation.
- **The Garbage Collection** that is performed through the GC. The Garbage Collection keeps track of all objects in the Java heap and safely removes unreferenced objects.

The JVM specification allows several different JVM implementations to exist for both open-source and proprietary purposes. Until today, the most popular implementation is the OpenJDK HotSpot JVM, and the vast majority of JVM implementations originate from it. The most notable alternatives to HotSpot are GraalVM [WWW+13], Eclipse OpenJ9 [Ecl] and MaxineVM [Mat08].

2.2.3 Optimising Compilers

JVM relies on high-performance application virtualisation to ensure cross-platform portability. Therefore, it employs optimising compilers to move slow architecture-independent bytecode interpretation to fast specialised architecture-dependent executables. For instance, the Java HotSpot VM has employed a tiered compilation approach that uses one of the two just-in-time (JIT) compilers, the client compiler [KWM+08] and the server compiler [PVC01a], also called C1 and C2, respectively. However, the state-of-the-art JIT compiler is Graal [ZSC13], an aggressive metacircular Java JIT compiler written purely in Java. This optimising compiler uses profiling information to guide speculative optimisations that yield better performance when speculative assumptions are correct. Although, if these become invalid under some circumstances, a deoptimisation occurs to ensure the continuation of the execution by migrating the execution to the interpreter from a specific stack pointer and bytecode index.

2.2.3.1 Graal Compiler

Graal [WWS10, WWW+13] is a high-performance optimising JIT compiler for the JVM platform implemented in Java to complement the existing JIT compilers (i.e., C1 and C2) in the HotSpotVM. The JVM bytecodes are passed to the Graal compiler, which builds its intermediate representation, the so-called Graal-IR [DWS+13]. The graph is then optimised through a tiered-compilation process, resulting in the generation of the architecture-specific code at the end.

Graal provides a seamless way of integration with the JVM through the JVM compiler interface (JVMCI) [Ros]. Through JVMCI the process of interchanging compilers in the HotSpotVM is more straight forward. Therefore, the standardised tiered compilation of the HotSpotVM C++ based compilers, the C1 and C2, can be replaced with the Graal compiler. JVMCI installs the final and optimised machine code in the VM's code cache. Therefore, other components and features of the HotspotVM, such as the bytecode interpreter and the garbage collector, can work uninterrupted.

Graal IR: Graal IR [DWS+13] is the an intermediate representation for a Java just in time (JIT) compiler written in Java. It uses nodes representing control-flow and data-flow dependencies in a Sea of Nodes [CP95] style. This intermediate representation is a directed graph with nodes based in the single static assignment (SSA) form [LG99]. SSA form guarantees that each variable is assigned only once across the graph, and therefore, numerous dynamic optimisations, such as canonicalisation and constant propagation, can leverage the single assigned variables.

Graal IR uses two types of nodes, for both control-flow and data-flow, floating and fixed nodes. Fixed nodes are associated with control flow nodes. Control flow nodes are always connected with their successors. For instance, nodes for preserving the structure of the graph, such as `LoopBegin`, `IFNode` and `InvokeNode`. In contrast to the control-flow nodes, floating nodes can be data flow nodes connected with their inputs, and there is no strict schedule for them. Therefore, during optimisation, dynamic optimisations can be performed while nodes associated with specific operations, such as memory accesses, can be scheduled into different basic blocks of the graph. For example, `ParameterNode`, `IndexedStoreNode` and `IndexLoadNode` along their corrensponding inputs.

The front-end of the compiler is responsible for turning the source representation, e.g., bytecode, into the Graal IR. The component repsponsible for this tasks is the `Graphbuilder`. It parses the bytecodes, as well as profiling feedback gathered by the interpreter into an IR graph.

Listing 2.1 showcases a Java method calculating the Fibonacci sequence up to an input upper bound by recursively calling the same method. Compiling the above code with the `javac` compiler results to the bytecodes in Listing 2.2. For the same example, the IR graph corresponds to a Control Flow Graph (CFG) as illustrated in Figure 2.5.

Listing 2.1: Example of Java method recursively calculaing a Fibonacci Sequence.

```
public static int fib(int number) {
    if (number <= 1) {
        return number;
    } else {
        return fib(number - 1) + fib(number - 2);
    }
}
```

Listing 2.2: Java bytecodes for the method `fib` presented in Listing 2.1

```
public static int fib(int);
Code:
   0: iload_0
   1: iconst_1
   2: if_icmpgt     7
   5: iload_0
   6: ireturn
   7: iload_0
   8: iconst_1
   9: isub
  10: invokestatic  #3                  // Method fib:(I)I
  13: iload_0
  14: iconst_2
  15: isub
  16: invokestatic  #3                  // Method fib:(I)I
  19: iadd
  20: ireturn
```

As shown in Figure 2.5, a Graal IR graph always starts with a `Start` node. Then, depending on the structure of the input code, different node types can be attached to the graph, while keeping different associations between them. For instance, in the above example, control flow nodes, such as an `If` node are always succeeded by two `Begin` nodes to represent the true and false paths on this branch. The same control flow node

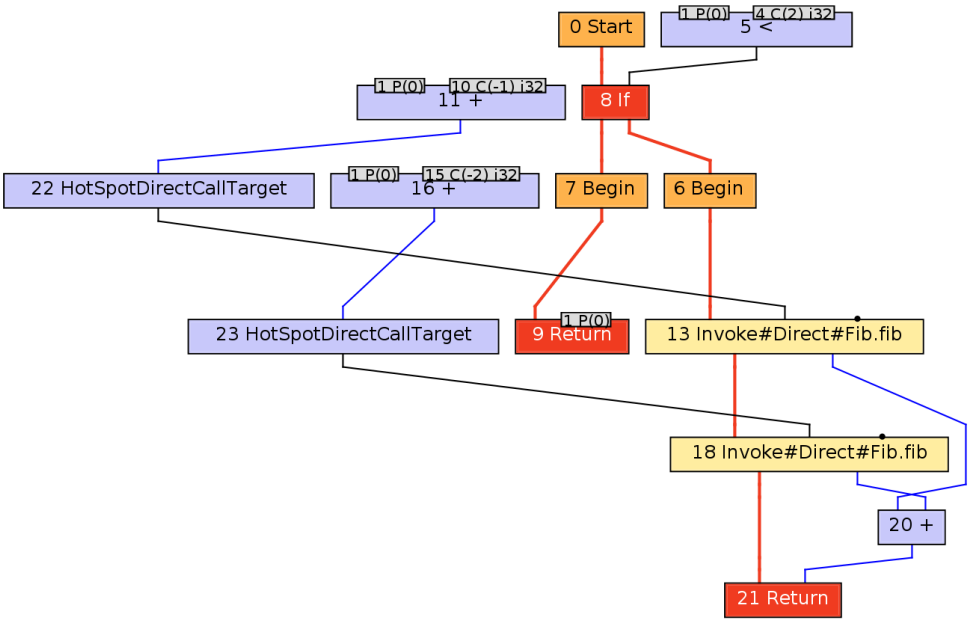


Figure 2.5: Graal IR for the Fibonacci sequence of method in Listing 2.1 obtained through the Graal graph builder of the bytecodes displayed in Listing 2.2.

has as an input the result of "less than" comparison between a constant (i.e., (C(2))) and an input object to a method, namely as a `Parameter` node. Also, in this example there are three instances of `Add` nodes; two for the values dynamically returned through the invocation of the same method during run-time.

Snippets: Graal also provides a utility to ease the implementation and lowering of architecture-dependent optimisations into the IR. The lowering process enables the compiler graph that still preserves a structure, offered through a *high-level* language, to be *lowered* to a representation that encapsulates architecture characteristics. This utility called Snippets [SWU+15] and is a improvement of C1X's templating mechanism, XIR[TWSC10]. Snippets provide an infrastructure to express low-level machine-oriented optimisations with a high-level structure. Optimisations are written as static Java methods, compiled into Graal IR during run-time, and then injected into the original compiler graph at the corresponding basic blocks. This process of inserting low-level semantics into the graph early on allows the exploitation of further optimisations, such as canonicalisation or virtualised memory accesses.

As shown in Figure 2.6, snippets are getting inserted into the graph during the lowering process from the high compilation tier to the mid compilation tier. During this process, the lifecycle of a snippet consists of three steps: Preparation, Specialisation,

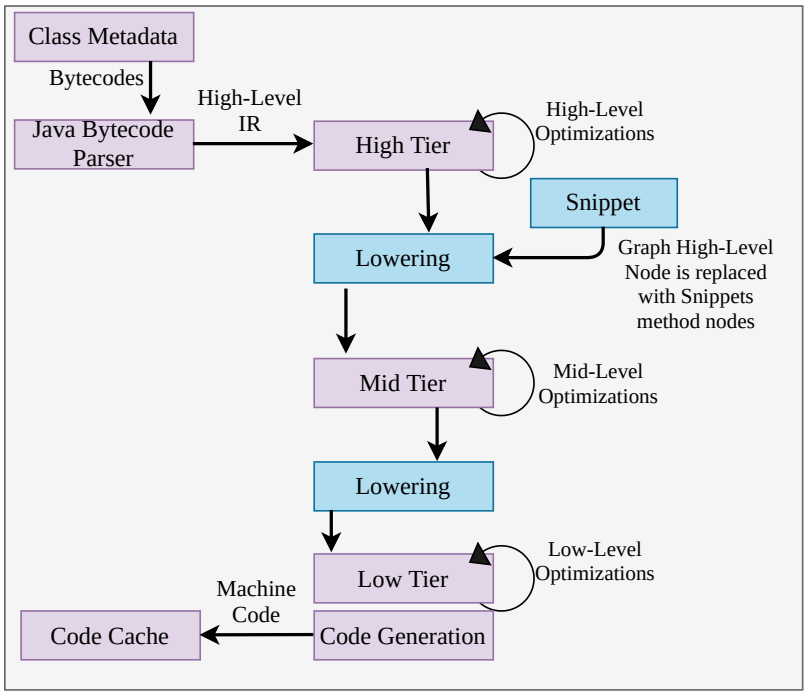


Figure 2.6: High-level overview of the snippets lowering process.

and Instantiation. Preparation processes a snippet like a regular Java method with its bytecodes parsed into an IR followed by several transformations through optimisation phases. Specialisation aims to reduce the graph size by eliminating source code guarded by constant parameters. The specialisation step is ensured by applying various optimisation phases into the newly created IR. Finally, during instantiation, the original high-level target code is replaced by the specialised snippet graphs.

Listing 2.3: Example of AArch64 integer division Snippet in GraalVM

```
@Snippet
static int idivSnippet(int x, int y,
    @ConstantParameter boolean needsZeroCheck) {
    if (needsZeroCheck) {
    zeroCheck(y);
    }
    if (x == Integer.MIN_VALUE && y == -1) {
        return Integer.MIN_VALUE;
    }
    return safeDiv(x, y);
}
```

As listed in 2.3, snippets can be used to address the integer division in a 64-bit ARM architecture (i.e., AArch64). Therefore, this infrastructure allows architecture-dependent operations to be expressed, specialised, and inserted to the original compiler graph without the need to use low-level primitives, such as inline assembly.

2.3 Heterogeneous Managed Runtimes

The advent of heterogeneous computing allowed modern computing systems to achieve high performance and moderate power consumption. Through this new computing norm, programming models have emerged while sacrificing simplicity. Hardware features are exposed to the user space to allow high performance. For instance, the OpenCL and CUDA parallel programming models previously discussed (Section 2.1.2) highly depend on features, such as managed memory allocations and dynamic typing. Thus, non-expert developers familiar with languages managed by the JVM cannot adapt heterogeneity in their programming routines.

Developers are willing to use heterogeneous hardware only if current programming practices do not change drastically. Managed runtimes to integrate this kind of programming environment need to evolve to provide the means to express heterogeneity. Therefore, one can strive towards Heterogeneous Managed Runtimes to see an intersection between conventional programming and heterogeneous hardware.

Clarkson *et al.* [CFP+18a, KCR+17a], state that in order for Heterogeneous Managed Runtimes to be practical they need to address several requirements. These requirements are: *Programmability*, *Transparency*, *Adaptability*, *Device Portability* and *Performance Portability*.

In this thesis, an experimental platform that satisfies the criteria mentioned above is needed. Therefore, it uses, the TornadoVM [FPZ+19, CFP+18c] framework as an experimental platform.

2.3.1 TornadoVM

TornadoVM [FPZ+19, CFP+18c, Cla19] is an extension to GraalVM that allows programmers to run Java programs on heterogeneous hardware automatically. Figure 2.7 illustrates a high-level overview of the TornadoVM framework that consists of three individual components:

TornadoVM API: The API exploits loop-parallelism, and it targets data-parallel

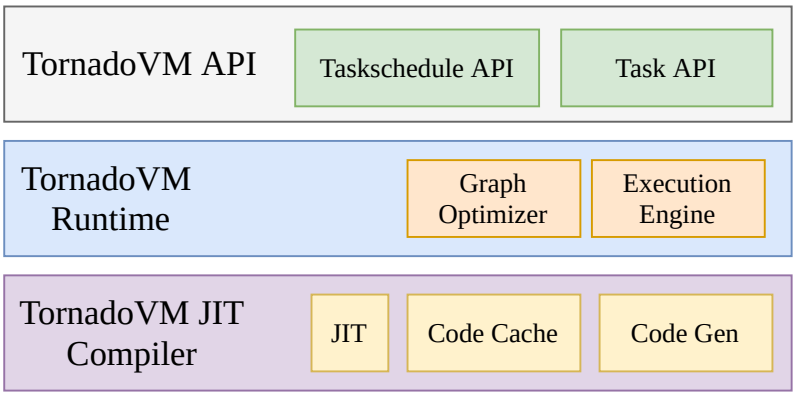


Figure 2.7: A high-level overview of the TornadoVM API, runtime and JIT compiler.

```
1  public class VectoAdd {
2   public void add(int[] a, int[] b, int[] c) {
3      for (@Parallel int i = 0; i < c.length; i++) {
4          c[i] = a[i] + b[i];
5      }
6   }
7   public void computeVadd(int[] a, int[] b, int[] c) {
8      TaskSchedule ts = new TaskSchedule ("s0")
9          .task("t0", this::add, a, b, c)
10         .streamOut(c)
11         .execute();
12  }
13 }
```

Listing 2.4: Example of the TornadoVM API to accelerate a simple Vector Addition between two primitive arrays.

workloads with minor modifications to an existing program. Developers do not need to take into account fine-grained architectural features neither to re-write their existing Java code from scratch.

Listing 2.4 showcases a simple vector addition written in Java with the TornadoVM API. In line 2, there is a method called *add*, which takes two primitive integer arrays and stores their sum into a third integer array. The only modification lies in line 3, where this loop is marked as parallel with the `@Parallel` annotation. Note that TornadoVM uses loose parallel schematics. Thus, in cases that the annotated loop propagates or carries dependencies, then the compiler will try to compile it, it will fail, and the execution will divert to normal JVM execution.

In addition, apart from the modification required to the target method, one needs to propagate this method into the TornadoVM framework explicitly. This is done by

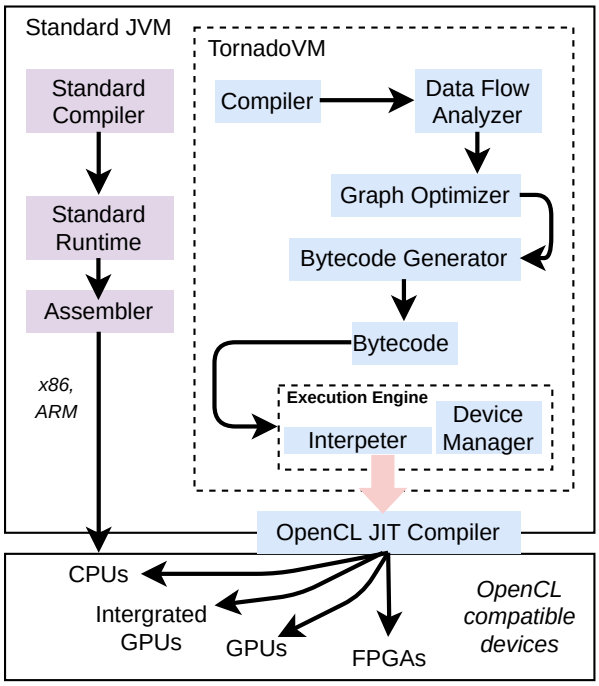


Figure 2.8: Overview of the TornadoVM heterogeneous programming framework.

instansiating a `TaskSchedule`. In line 8 there is a declaration for a `TaskSchedule` associated with the above created `Task` for the `add` method. The `TaskSchedule` can have an arbitrary number of tasks with-or-without dependencies between them. Each `task` declaration has an internal identifier (i.e., t0), the method handle, and the arguments needed for the given method. The `TaskSchedule` API offers an explicit `streamOut` call to indicate a variable that needs to be returned from the device to the host after execution. Finally, the `execute()` call corresponds to the beginning of the offloading process, including the compilation of the task into OpenCL code.

TornadoVM Runtime: Figure 2.8 showcases how the TornadoVM runs inside a standard JVM (e.g., the HotSpot JVM [PVC01b]). The TornadoVM runtime is responsible for orchestrating execution in a managed manner by analysing the dependencies between tasks of a `TaskSchedule`, to minimise the overhead of data transfers. Besides, TornadoVM implements and uses its own interpreter along with a set of specialised bytecodes dedicated for heterogeneous execution. These specialised TornadoVM bytecodes, along with a brief description of their purpose, are outlined in Table 2.2. Note that TornadoVM treats execution at a `TaskSchedule` granularity. Therefore, `TaskSchedules` having single or multiple tasks can only target a single device for execution even when systems are equipped with several heterogeneous devices.

TornadoVM provides a memory management scheme to explicitly control data

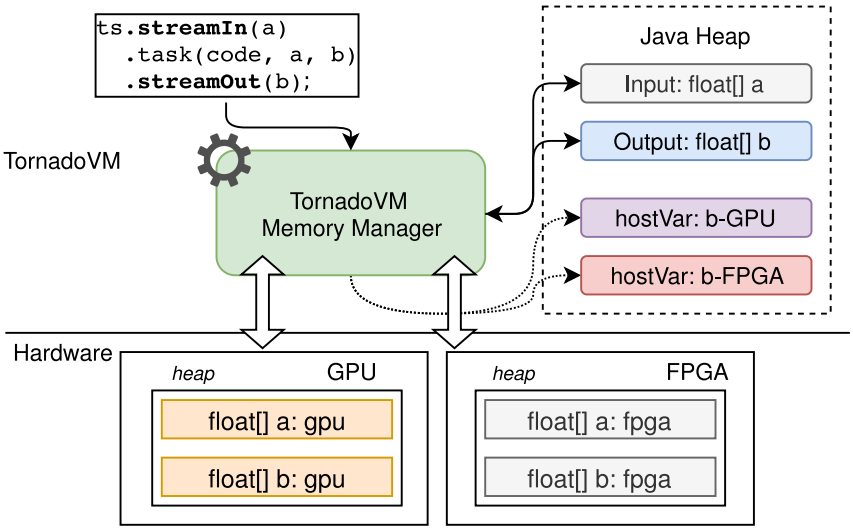


Figure 2.9: TornadoVM Memory Management Scheme.

allocation and movement between the host JVM and the target device. The memory manager is responsible for maintaining consistency of variable references between the host JVM and the device. Figure 2.9 depicts a high-level overview of this scheme. The allocation is performed through Java Native Interface (JNI) calls to the OpenCL functions for allocating buffers onto the device. Also, the memory management scheme pre-allocates a memory region on the global memory of the accelerator. The size of this allocation reflects the maximum memory capacity of the device, or the maximum size allowed by the OpenCL standard. Therefore, the memory manager becomes solely responsible for transferring data between the host and the target memory regions while ensuring memory consistency at run-time.

TornadoVM JIT Compiler: The JIT compiler uses and extends the Graal compiler and the Graal IR. The JIT compiler is responsible to handling the generation of OpenCL

Table 2.2: List of the TornadoVM bytecodes along with a description.

Bytecode	Operands	Description
BEGIN	<context>	Creates a new parallel execution context.
ALLOC	<context, BytecodeIndex, object>	Allocates a buffer on the target device.
STREAM_IN	<context, BytecodeIndex, object>	Copies an object from host to device.
COPY_IN	<context, BytecodeIndex, object>	Alloc & copies an object from host to device.
STREAM_OUT	<context, BytecodeIndex, object>	Copies an object from device to host.
COPY_OUT	<context, BytecodeIndex, object>	Alloc & copies an object from device to host.
COPY_OUT_BLK	<context, BytecodeIndex, object>	A blocking COPY_OUT operation.
LAUNCH	<context, BytecodeIndex, task, Args>	Executes a task, compiling it if needed.
ADD_DEP	<context, BytecodeIndices>	Adds a dependency between labels.
BARRIER	<context>	Waits for all previous bytecodes.
END	<context>	Ends the parallel execution context.

code for the input methods annotated with the `@Parallel` for a given TaskSchedule. These methods are compiled into Java bytecodes through `javac`, and then into Graal IR with the Graal's graph builder. In addition, from this early graph, the TornadoVM compiler evaluates the parallel dimensions of a given task by analyzing the induction variables and loop bound of the annotated methods.

TornadoVM augments the tiered compilation of Graal with several optimising phases to specialise the IR to target heterogeneous code generation. For instance, an IR optimisation phase, such as *Parallel Scheduling* aims to transform the loop induction variables to use OpenCL thread identifiers, such as the `getGlobalID` by inserting new graph nodes into the IR. Therefore, through this process and entirely seamlessly from the user, the `@Parallel` annotation hints the compiler to exploit loop parallelism for Java applications.

Finally, the JIT compiler, instead of machine code, generates OpenCL or PTX code directly from the optimised graph. However, the final executable in a binary format is obtained by compiling the generated code with the compiler that each device vendor provides. Therefore, in this way, more device-specific optimisation can be effortlessly achieved. Then, the execution engine that lies in the runtime of TornadoVM directly handles the executable through JNI calls to the corresponding OpenCL function, such as `clCreateProgramWithBinary`.

Dynamic Reconfiguration [FPZ+19]: TornadoVM offers an infrastructure to maximise the potential single device performance. To do so, a dynamic reconfiguration feature is provided, along with some execution plan policies. Through exhaustive exploitation of all the available devices, depending on the selected policy, TornadoVM decides and reuses the best device, until the input method is altered. The different policies can influence the device selection based on criteria, such as minimal latency or peak end-to-end performance.

2.4 Machine Learning Modelling

Machine learning (ML) is the research area that focuses on developing algorithms capable of automating specific processes and tasks. In more detail, a machine learning algorithm can learn how to perform a given task by iterating through sample data fed to it as input. This sample data is usually called a *training set*, and every single observation in the training set is called a data point.

Formally, a machine learning algorithm can be described as follows: *A computer*

program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E [Mit97]. Depending on the type of the training set fed to or equivalently experienced by the ML system, the learning process can be divided into four different types: Supervised, Unsupervised, Semi-supervised, and Reinforcement learning. Among these types, supervised and unsupervised learning are the most commonly used and they will be further explained in Section 2.4.1.

The problems that can be solved using ML are usually highly complex for humans to process or are repeated with some variations. Depending on how the machine learning system is expected to process a given training set, the tasks can be further divided into subcategories. Among these categories, some of the most common are regression, classification and clustering tasks that will be presented in more detail in Section 2.4.2. This thesis uses classification to address the performance challenge which is presented in Chapter 6.

The quality of a given ML model is evaluated through specific performance metrics. Such performance metrics are evaluated by using data unseen by the algorithm, usually called a test set. In contrast to the training set, assessing the performance on the test set reflects the objective performance of the model in real-world scenarios.

2.4.1 Supervised vs Unsupervised Learning

Supervised learning requires a dataset containing several data points and various features [Kot07]. These features are several variables that can influence a specific outcome or behaviour. Also, each data point is accompanied by a target value or label. The training set format can be represented as a set of pairs consisting of an input object. Usually, a vector containing the feature values for the specific data points and the associated output values is also called a supervision signal. It must be noted that the desired output values can be either numerical or they can fall into specific categories.

In supervised learning, the goal of the algorithms is to learn to associate the input training set with the given target value or label. In other words, it aims to identify a mapping function between the input features and the target variable. This inferred function should also perform adequately to previously unobserved data, i.e., to have generalisation ability.

For an algorithm to generalise beyond the examples that were explicitly trained with, it is required that specific conditions hold. One requirement for the algorithm is to model only the significant underlying patterns in the data and not the noise. Otherwise,

it would memorise the training set, leading to a poor generalisation to any unseen data. This problem is called overfitting [BA10]. However, the model to be able to capture the given data structure, despite its potentially high mathematical complexity, without oversimplifying, it has the same importance. Otherwise, its overall performance will be rely on both the training and test sets withoun be adapting to unobserved data.

On the other hand, unsupervised learning requires some unlabelled data points and various features [HS99]. In these cases, the usual goal is to reveal any useful properties of the dataset's underlying structure without human intervention. Unsupervised algorithms are usually employed for tasks, such as clustering and principal component analysis (PCA) [Jol86].

2.4.2 Classification Problems

Classification tasks are commonly solved while using supervised learning. The goal, in this case, is to associate a given data point with a class label. In more detail, the algorithm learns to associate any given an example from the training set X with a specific category, described by the target variable y. Formally, this is achieved by producing a function f : Rn $\rightarrow$ 1, . . . , k, where k is the number of available classes. However, instead of directly outputting class labels, in many cases, the inferred function f outputs a probability distribution over the available classes. The assigned probabilities can be then turned into class labels using a process named threshold-moving or threshold selection [HM13].

Threshold selection is the process of defining the most suitable decision threshold, also called decision boundary, that will govern the conversion of the predicted probability into class labels. A common case is this of a binary classification problem, i.e., a classification with only two target classes. In this case, the default threshold is set at 0.5 so that output probabilities less than 0.5 are mapped to class 0 while those greater than 0.5 are mapped to class 1. However, under certain circumstances, this default threshold may not reflect the optimal interpretation of the output probabilities. This can be the case for many reasons, including skewed class distribution and increased misclassification cost of one type over another. To handle these cases, one can adjust the threshold based on domain expertise or by optimising for specific evaluation metrics, such as F1-score, which will be further analysed in Section 6.3.4.

2.5 Summary

This chapter presented the terminology needed for understanding the rest of this thesis. Briefly, it presented the computer architecture of the Graphics Processor Units (GPUs), the Field Programmable Gate Arrays (FPGAs), Multicore Central Processing Units (CPUs), as well as the OpenCL parallel programming model. Also, it outlined the key concepts to understand managed languages and their runtimes with particular attention to the Java language and the JVM. It focused on the workflow of a metacircular JIT compiler, such as the Graal compiler, along with techniques to compile the Java bytecodes to efficient machine code at run-time. In addition, this chapter introduced TornadoVM, a heterogeneous managed runtime that is extensively used in the scope of this thesis. Finally, the last section briefly explained the fundamental concepts required to model and solve a problem by employing machine learning techniques.

The following chapter presents the state-of-the-art research work that is associated with the scope of this thesis.

Chapter 3

Related Work

3.1 Introduction

The previous chapter has introduced the hardware and software concepts necessary in order to understand the ideas realised through this thesis. This work spans vertically into several research directions. Therefore, this chapter narrows down the scope of the related state-of-the-art work. It analyses techniques, technologies, and approaches that raise the level of abstraction for heterogeneous hardware to make it accessible from high-level languages. This analysis highlights the gaps in the current state-of-the-art that the contributions of this thesis aim to fulfil.

Section 3.2 discusses various approaches for integrating FPGA execution in high-level languages. Section 3.3 highlights the related work on compiler optimisations and techniques to automatically exploit data locality for GPU code generation. Finally, Section 3.4 presents several research contributions aiming to utilise multiple heterogeneous devices by scheduling single or multiple applications onto them.

3.2 FPGA Execution of Managed Languages

FPGA acceleration of managed languages is a well-studied research topic with a broad range of approaches widely available. In the scope of this thesis, the prior work of managed languages (e.g., Java, Python, C#) that utilise FPGA hardware can be classified into the two categories illustrated in Figure 3.1. The first category targets programming languages that interface statically with pre-compiled FPGA designs, while the second category includes languages that generate FPGA code dynamically.

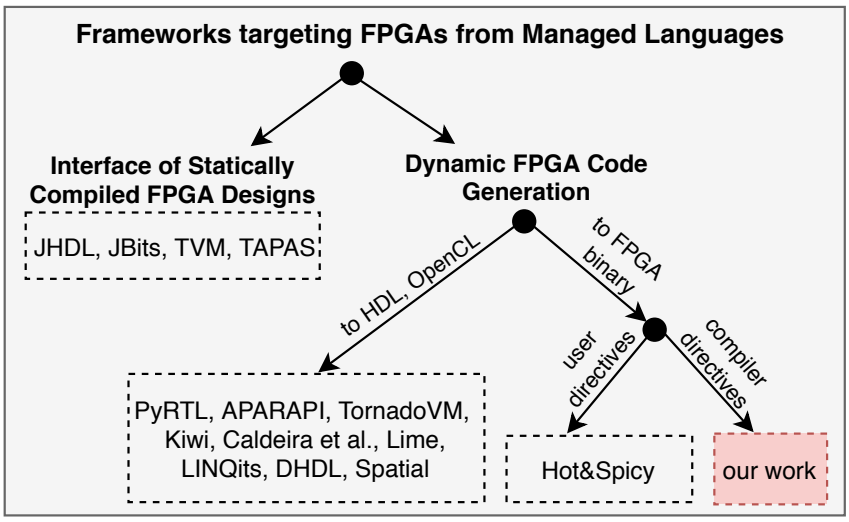


Figure 3.1: Classification of the state-of-the-art frameworks that target FPGA hardware from managed languages.

3.2.1 Interfacing with static FPGA designs

Bellows and Hutchings [BH98] introduced the JHDL framework, which describes how a Java interface should be constructed for accessing the FPGA hardware by calling existing bitstreams. Guccione *et al*. [GLS99] presented the JBits API that requires hardware knowledge to use hardware accelerators. The API is purely written in Java; however, one can only interact and construct designs for which circuitry components are already available in the software development environment.

Additionally, several frameworks target applications written with domain specific languages (DSLs) that provide bindings to or are built on top of other managed languages. These frameworks compile the application written with a DSL into hardware units that can be accelerated at run-time onto FPGAs. Moreau *et al*. [MCC18] extended the TVM [CMJ+18], an optimising compiler for deep learning applications, to utilise FPGAs. This framework targets multiple models from Python-based machine learning frameworks, such as TensorFlow [AAB+15] and Keras [C+15]. Margerm *et al*. [MSG+18] presented the TAPAS framework that analyses task dependencies in the compiler graph and generates a data-flow processing unit transparently to be executed on the FPGA.

3.2.2 Dynamic FPGA code generation

Several frameworks that enable FPGA-based acceleration of managed languages through dynamic code generation exist. Clow *et al.* [CTD+17] presented the PyRTL framework which compiles Python programs into Verilog. However, it requires programmers to be familiar with specific design practices and hardware primitives. Segal *et al.* [SMCW] extended the GPU capabilities of a Java-based framework, APARAPI [AMD16], to run on OpenCL-compatible FPGAs. However, the proposed framework did not support any automatic optimisation phases and several kernels required manual intervention to be synthesised for FPGA boards. Greaves and Singh [SG08] presented the Kiwi library that exposes various custom attributes to the programmers and generates Verilog from the input applications written in C#. Caldeira *et al.* [CPB+18] proposed a framework that compiles Java programs into Verilog. However, further work is required to imprint the resulting Verilog code into the FPGA available on the Intel HARP platform. Also, Skalicky *et al.* [SMSF18] proposed Hot&Spicy to compile code written in a subset of Python into HLS C code and transparently invoke the Xilinx SDSoC HLS tool to produce the FPGA binary. In this case, the user needs to annotate the input program with several vendor-specific pragmas to trigger FPGA optimisations by the HLS tool.

Furthermore, several approaches have proposed other DSLs (e.g., Lime [ABCR10], Language Integrated Query (LINQ) [CDL13], Delite Hardware Definition Language (DHDL) [KPZ+16], Spatial [KFP+18]) to generate code for various FPGA hardware description languages [KB16]. Auerbach *et al.* [ABCR10] presented the Lime language and framework within the streaming domain that compiles Java programs to C and Verilog. Chung *et al.* [CDL13] proposed the LINQits framework that allows various Big Data workloads to be compiled by the Dandelion [RYC+13] compiler and accelerated on FPGA hardware. However, LINQits does not support automatic HLS compilation and requires programmers to introduce the HLS directives. Koeplinger *et al.* [KPZ+16] also required users to write their program into the DHDL language, which then compiled into MaxJ. MaxJ is a low-level Java-based language that allows the generation of hardware for the Maxeler platform using the MaxCompiler. Besides, Koeplinger *et al.* [KFP+18] proposed the Spatial language as a compiler extension to the DHDL, thereby allowing developers to gain more control over the memory hierarchy from the programming language.

Moreover, some frameworks use the abstraction that compiler intermediate representations (IRs) provide to dynamically generate HLS-compatible source code. Del Sozzo *et al.* [DBAS18] introduced FROST, a unified backend for targeting FPGAs

Table 3.1: Taxonomy of the state-of-the-art frameworks that target heterogeneous execution from Java

Frameworks	**Code Generation**	**Run-time Optim.**	**HLS Comp. Mode**	**Hardware Platforms**
TVM [MCC18]	static	No	No	FPGA Pynq SoC
MaxCompiler [Max11]	dynamic	No	online	Maxeler Platform
Aparapi [SMCW]	dynamic	No	offline	AMD GPUs, FPGAs
Caldeira et al. [CPB+18]	dynamic	Yes	online	Intel Harp FPGAs
JOCL [FPZ+19]	dynamic	No	offline	GPUs
TornadoVM [FPZ+19]	dynamic	No	offline	CPUs, GPUs, FPGAs
TornadoVM + FPGA extensions	dynamic	Yes	online, offline, emulation	CPUs, GPUs, FPGAs

through DSLs, such as Halide [RKBA+13] and Tiramisu [BRR+19]. FROST provides its own IR, graph optimiser with FPGA-oriented passes, and a scheduling co-language to allow users to specify optimisations. Izraelevitz *et al*. [IKL+17] presented FIRRTL, a Flexible Intermediate Representation for register-transfer level (RTL) code. FIRRTL is integrated with Chisel [BVR+12], a hardware design language that facilitates advanced circuit generation and design reuse for ASIC and FPGA digital logic designs. Also, it transforms target-independent RTL into design-specific RTL through several optimisation steps, such as simplifying transformations, analyses, optimisations, instrumentation, and specialisations.

3.2.3 Java Execution on FPGAs: Spotting the Gap

Table 3.1 summarises the currently available frameworks that enable FPGA acceleration of Java programs or Java-based DSLs. The selection of frameworks is analysed based on the following four criteria:

1. **Code Generation:** The ability to generate parallel code at compile-time (statically) or during runtime (dynamically). For example, the code generation from Java to OpenCL or Verilog during run-time is classified as dynamic.

2. **Run-time optimisations:** The ability to automatically specialise code for the target device at runtime without user intervention (including code annotations).

3. **HLS Compiler Mode:** The ability to perform an online or offline compilation from the generated code to the final FPGA bitstream via the HLS compilers.

4. **Hardware Platforms:** The supported hardware platforms for FPGA acceleration.

Beyond the state-of-the-art: The novel FPGA-aware JIT compilation system integrated with a heterogeneous managed runtime that is presented in this thesis in Chapter 4 advances the related work in the following aspects:

- It automatically and dynamically compiles Java programs onto optimised FPGA-compatible code.
- It completely omits the need to use hardware-specific directives from programmers. Instead, this functionality is transparently enabled by augmenting the Graal IR and the existing optimising compiler.

3.3 Optimising Compilers for GPU Code generation

Another essential aspect of bridging the performance gap between heterogeneous hardware and managed languages is optimising the compiler for GPU code generation. In this section, the related work that targets GPU memory optimisations is presented. The related work is classified into two distinct groups. The first group in Section 3.3.1 describes approaches that exposing GPU features to a wide range of high-level programming languages while preserving the need for hardware understanding. The second group in Section 3.3.2 focuses on various memory transformations at the compiler level to omit any necessary understanding of the underlying hardware from the developer.

3.3.1 Exposing GPU Features into Programming Languages

Dynamic compilation allows a broad range of applications to attain high performance during execution by integrating profiling information during compilation. Thus, seamless GPU execution in the context of dynamically compiled languages, such as Java, is a well-researched topic. Several frameworks [AMD16, FPZ+19] provide solutions to exploit GPU acceleration. Aparapi [AMD16] and TornadoVM [FPZ+19] are Java-based frameworks that dynamically compile Java bytecodes to OpenCL and PTX code. Aparapi provides direct integration and native Java code execution on GPUs. However, its programming aspects preserve the need for a hardware-oriented approach as it is imitating the OpenCL programming approach. Therefore, it exposes specific language constructs for memory allocation (i.e., local memory) and memory synchronisation (i.e., barriers) that programmers must explicitly use [AMD16]. On the contrary, TornadoVM

generates high-level bytecodes to abstract programmers from the GPU programming model. However, it does not automatically exploit fine-grain memory, but it exposes low-level features to developers. Moreover, IBM J9 [IHKS15] is another example of a JIT compiler for GPU offloading, but it exclusively compiles Java streams to CUDA-PTX code. The only memory optimisation supported by the IBM J9 is the placement of read-only data to read-only caches. The most recent framework is the grCUDA [RM20] which provides a polyglot framework on top of the GraalVM [WWW+13] and Truffle [GSS+15] that enables GPUs acceleration.

Additionally, several parallel programming frameworks exist [SRD16, JTS19b, RLSD16, SBL+14, FRSD15, DCR+12, SKG11, CMJ+18, KFP+18] that enable the compilation of domain-specific languages on GPUs. Lift [SRD17, HSS+18] extended its original data-parallel primitive types to accommodate loop tiling (e.g., `slide, pad`) and its low-level OpenCL with local memory (e.g., `toLocal`) allocation to generate high performance code for stencil computations. Ragan-Kelley *et al.* [RKBA+13, RKAS+17a, RKAS+17b] introduced Halide, a domain-specific language (based on C++) for executing high-performance image processing code on GPUs. Halide has its own IR and an optimising compiler, both of which are highly tailored to perform optimisations for domain-specific applications. The main domain that it targets is image processing applications.

Beyond the state-of-the-art: The key differentiation of this thesis with prior work is the approach that Chapter 5 presents. This approach automatically exploits the GPU memory hierarchy without exposing any specific language primitives to the developers. In addition, it targets GPU architectures from various hardware vendors while using information during run-time to customise optimisations and code generation.

3.3.2 Compiler Techniques for Memory Transformations

Verdoolaege *et al.* [VCJC+13] used polyhedral models to automatically transform C code to CUDA while utilising shared memory and loop tiling. Similarly, Bondhugula *et al.* [BRS07] proposed PLUTO, an automatic loop nest paralleliser to exploit data locality via allocation in the shared memory of GPUs. Also, Grosser *et al.* [GCK+13] have extended the polyhedral models available in PLUTO with support for loop splitting targeting stencil workloads. PolyJIT proposed by Simburger *et al.* [SAGL18] combines polyhedral optimisations with multiple kernel versions available at run-time; a technique that poses significant overhead during code generation. Some research studies [BBK+08, BRR+19, KKRS14, RKH+11] target loop tiling optimisations

and code generation for affine loops targeting GPU code generation. Moreover, Di *et al*. [DYS+12] proposed an algorithm to improve tiling hyperplanes by using dependency analysis, while Cohen *et al*. [CGK+13] developed a polyhedral-based parametric scheme that uses run-time exploration of partitioning parameters.

Moreover, several non-polyhedral-based compiler approaches address the same issue. Kim *et al*. [KSRT+19] presented an approach to map tensor contractions directly to GPUs. This is achieved with the use of shared memory through a parametric code generation strategy that leverages a cost model to perform efficient data movements. Yang *et al*. [YXKZ10] introduced an optimising source to source compiler for C programs that exploits many memory optimisations, such as converting non-coalesced accesses to coalesced for reducing memory aliasing, vectorisation of memory accesses, and tiling with shared memory. Additionally, Hagedorn *et al*. [HLK+20] proposed Elevate, a new functional language to express various optimisations, such as vectorization, loop tiling, and loop splitting.

Beyond the state-of-the-art: The approach proposed in Chapter 5 in this thesis differs from the prior work presented above as it highlights a practical trade-off between compilation time and achieved performance. Therefore, this alternative approach is more suitable for interpreted and dynamically compiled programming languages. In addition, it presents its practical capability by displaying how it can be integrated into a production-ready compiler, such as the Graal [DWS+13] compiler.

3.4 Dynamic Application Scheduling on Heterogenous Hardware

Orthogonally to the challenges of generating code for heterogeneous hardware is to supply the means needed for efficient device scheduling. Therefore, the following related work focuses on multi-device utilisation through various scheduling techniques. The available research work that is closely aligned with the scope of the work that Chapter 6 presents has been classified into three groups. Section 3.4.1 discusses the first group of approaches that rely on non-machine learning techniques to perform task scheduling. On the contrary, Section 3.4.2 discusses approaches that use machine learning to improve the efficiency of the task scheduling process in systems with multiple devices. Also, Section 3.4.3 presents a third group that elaborates on approaches that allow single tasks to be executed on multiple devices while using machine learning.

3.4.1 Non-Machine Learning Multi-Task Scheduling

There have been many frameworks focusing on single or multi-task scheduling for standalone or partitioned applications purely written in OpenCL, such as VirtCL [YWTC15], SnuCL [KSL+12], PySchedCL [GSK+20], FluidiCL [PG14], MultiCL [APBcF16], EngineCL [NBB19] and SOCL [HBD+13]. However, all the aforementioned frameworks still rely on hardware understanding and applications written in unmanaged programming languages.

Beyond the state-of-the-art: The approach proposed in Chapter 6 in this thesis differs from the above mentioned work as it enables concurrent execution and scheduling on heterogeneous hardware for Java applications. Also, the proposed approach does not expose hardware specifics to the user.

The only work that differs Parravicini *et al.* [PDAS20] use the grCUDA [RM20] polyglot API and employ a custom scheduling approach to allow multiple polyglot tasks to be scheduled on a single GPU at run-time. This work exploits pace-sharing and overlaps the time spent in transferring data with the execution, if possible. Also, this work relies on expressing computational kernels in a way to be interpreted into a directed acyclic graph (DAG) by the scheduler, thus it is shifting away from the programming model that they are expressed into.

Beyond the state-of-the-art: Although the above work can target the same programming language, Java, due to Truffle [GSS+15] interoperability provided by GrCUDA it differs from the work presented in Chapter 6. This thesis focuses on using existing Java methods using the TornadoVM API to execute on multiple devices from different vendors offering diverse performance and power characteristics.

In addition, similar scheduling techniques have been used for heterogeneous task graph/multi-thread scheduling on asymmetric multicore processors [JCBM16]. In this work, JVMs, such as Jikes RVM and HotSpot can be augmented with the ability to identify and schedule critical threads to small or big cores based on their requirements. Managed applications can be dynamically analysed as single threaded, non-scalable multi-threaded, scalable multi-threaded, and get scheduled accordingly.

Beyond the state-of-the-art: Although the above work can target the JVM and managed applications that require more efficient scheduling, it only targets single-ISA asymmetric multicore processors. This thesis differs by augmenting the JVM with the ability to strive for performance on heterogeneous devices with different ISAs, such as GPUs, low-power GPUs and FPGAs.

3.4.2 Machine Learning-based Multi-Task Scheduling

Troodon [KAA+19] is a load-balancing scheduling heuristic that classifies OpenCL applications as suitable for CPU or GPU execution, based on a speedup predictor. The Qilin [LHK09] compiler uses offline profiling to create a regression model for predicting the execution time of a selection of input applications taken from the CUDA Software Development Kit. Ogilvie *et al.* [OPWL15] introduced a low-cost predictive model for the automatic construction of heuristics that reduce the training overhead for executing on platforms that utilise CPUs and GPUs. Furthermore, Grewe *et al.* [GWO13] leveraged predictive modelling to influence the OpenCL code generation from OpenMP programs when speedups are predicted. Additionally, Chen *et al.* [CMJ+18] combined generic search with learning and benchmarking to find appropriate scheduling methods for execution on heterogeneous hardware, including CPUs, server GPUs, mobile GPUs, and FPGA-based accelerators. However, the supported scheduling mechanism is semi-automated, as the search space must be manually defined by a programmer for each algorithm like a template. Wen *et al.* [WWO14] demonstrated that the concurrent execution of OpenCL kernels can increase GPU utilisation and improve performance. To enable such performance improvement, they applied a decision tree-based prediction model to decide whether an application kernel should be scheduled individually or alongside other kernels. Baldini *et al.* [BFA14] use existing OpenMP applications and supervised learning to predict the potential GPU execution speedup among vendors. Brown *et al.* [BNS+21] presented a model that allows accurate predictions of speedups using a small set of features, while also being portable across Nvidia GPUs with different capabilities. Adams *et al.* [AMA+19] proposed a novel scheduling algorithm for the Halide programming language that targets image processing pipelines. Their model combines symbolic analysis with machine learning to predict performance.

Beyond the state-of-the-art: The contribution described in Chapter 6 of this thesis provides similar functionality with the above work. However, it differentiates from the above as it extends a standard JVM and allows applications written in Java to exploit concurrent execution on multiple heterogeneous devices.

3.4.3 Single Task Scheduling on Multiple-Devices

Other studies have combined predictive modelling with scheduling to allow a single task or application to be partitioned, and execute onto multiple devices. Kofler *et al.* [KGCF13] employed an Artificial Neural Network to dynamically partition a

given task into two parts, one that runs on a CPU and a second that runs on a GPU. This partitioning is achieved through the Insieme compiler [JPT+13] that transforms the code from a single kernel into multiple kernels. Grewe *et al*. [GO11] presented a system that combines a two-level predictor with supervised learning models (i.e., Support Vector Machines) to partition tasks into percentages of the input application for hybrid CPU-GPU execution. The model uses features extracted statically from the abstract syntax tree (AST) that LLVM [LA04] generates, and then these features were normalised to the specific data requirements of each input program. Also, Singh *et al*. [SPB+17] presented a runtime system that performed energy-efficient mapping and repartitioning of threads for each application between CPU and GPU cores of an multiprocessor system on a chip (MPSoC) while considering the applications execution time.

Beyond the state-of-the-art: Chapter 6 presents a contribution that differs from the afforementioned work as the prime focus is to provide the tools and techniques to enable the seamless and intelligent mapping of multiple applications instead of a single to be able to execute onto multiple devices.

3.5 Summary

The content of this thesis extends across several multidisciplinary research domains. This chapter outlined the state-of-the-art regarding the research areas that are intersect with this work. To achieve this, this chapter provided a classification of the related work in three distinct areas of interest: **(i) approaches that raise the level of abstraction for FPGA execution**, **(ii) compilers that exploit memory optimisations for GPU code generation**, and **(iii) novel frameworks and approaches that enable multi-device execution on heterogeneous systems**.

The following chapters discuss how this thesis augments the existing state-of-the-art of the aforementioned research domains. In particular, Chapter 4 explains in detail how FPGA execution can be integrated with heterogeneous managed runtimes, and yield higher performance against various platforms. Furthermore, it gives a detailed description of the key challenges when enabling such functionality, while providing a combination of runtime and compiler optimisations.

Chapter 4

FPGA Aware JIT Compilation for Managed Runtime Programming Languages

Modern computing systems integrate various hardware devices, such as CPUs, GPUs, and FPGAs, to offer high performance and energy efficiency. However, achieving high performance in such systems is considered a challenging task even for expert programmers. Programming FPGA devices requires a deep understanding of the computing hardware (e.g., reconfigurability) and familiarity with low-level Hardware Description Languages (HDL), such as Verilog [TM96] and VHDL [Ash08, Per93].

Several researchers from industry and academia have been trying to mitigate the steep learning curve of programming an FPGA by providing High-Level Synthesis (HLS) [IKL+17] and heterogeneous programming frameworks (e.g., OpenCL). These tools aim to facilitate and simplify the development on FPGAs, especially nowadays where they can be found on off-the-shelf Systems on Chip (SoCs) [CHL+17] and in data centres or cloud deployments [PCC+14]. However, in the computing world where the adoption of managed languages is dominant, their current support for FPGA execution is minimal. Even though several JIT compilers that target GPUs have recently emerged [FRSD15, PSFW12, ZLG12, FSSD17, AAA16], in the FPGA domain, such approaches cannot be directly applied due to lack of *performance portability* [ZSC13]. Therefore, it has become a necessity to expose low-level hardware primitives to the high-level programming model [SMSF18].

The focus of this chapter is to present a novel and practical approach that explores how to automatically specialise FPGA code and transparently integrate state-of-the-art

HLS tools into managed languages. The outlined solution is developed and evaluated in the context of TornadoVM [FPZ+19] (previously discussed in Section 2.3.1). The original framework has been extended by enabling it to generate and handle FPGA bitstreams automatically. This process eliminates the necessity of external invocation of HLS tools and manual binding of the executed FPGA code with the current execution flow of TornadoVM. The primary objective is to enhance managed languages with automatic FPGA code specialisation and execution without requiring the programmers to explicitly add any hardware-related code annotations.

Briefly, this chapter makes the following contributions:

1. It details the design of an open-source end-to-end toolchain capable of transparently compiling and running Java code on an FPGA by introducing a set of execution modes to the Java Virtual Machine (JVM). This approach enables developers with no FPGA expertise to harness the execution capabilities of FPGAs. This toolchain also provides seamless integration with emulation tools provided by the HLS vendors to enable fast prototyping through standard IDEs, such as the IntelliJ.

2. It presents the implementation of a compiler-based approach to automatically exploit parallelism in the IR-level and specialise the IR for OpenCL code generation targeting FPGAs. The introduction of new compiler phases augments and transforms the IR with FPGA-specific pragmas.

3. It presents the evaluation of the complete framework against Java benchmarks running on FPGAs, showcasing end-to-end speedups of up to 19.8× and 224× over multithreaded and sequential Java code, respectively. The implemented end-to-end framework results up to 13.82× faster execution compared to the execution on an integrated GPU through TornadoVM. Finally, it classifies the given selection of benchmarks based on their applicability for FPGA acceleration from managed runtime languages.

The rest of this chapter is organised as follows: Section 4.1 it tries to motivate the FPGA execution of Java programs. Then, Section 4.2 presents how functional integration of a heterogeneous managed runtime and an FPGA can be achieved through seamless integration with HLS. Moreover, Section 4.3 extends beyond the seamless integration and targets to address any performance issues by augmenting an industrial standard JIT compiler with FPGA-oriented optimisations. Finally, Section 4.4 discusses

the obtained performance gains, along with the produced compilation overheads and the applicability of the given benchmarks for FPGA execution.

4.1 Motivation: FPGA Performance for Unoptimised Auto-Generated OpenCL Kernels

As presented in Section 2.3.1, TornadoVM enables Java developers to write task-based programs that are automatically compiled and executed on heterogeneous hardware. Prior to this work, the original framework supported the execution mentioned above only for GPUs and CPUs. This limitation resulted from a different compilation and execution flow required for FPGAs compared to the devices stated above. One needs to manually compile the auto-generated OpenCL code through a vendor-specific HLS software, deploy the generated FPGA bitstream, and redirect the execution from TornadoVM to the FPGA. Also, the original framework did not perform any compiler optimisations specifically for FPGAs. The particularities of the aforementioned process prohibited the seamless FPGA execution from within TornadoVM.

Initial Support with Manual Intervention: To assess the potential performance benefits of executing Java programs on FPGAs through TornadoVM the following process took place. During this process, a *DFT* computation application was used. The reason for using this application as a use case is the fact that it exhibits data parallelism workload which is heavily relies on using function invoked directly from the Java Math collection. Initially, bitstreams for three different workload sizes, small, medium, and large were obtained. In all cases, the attained performance was slower than the single-threaded Java execution on CPUs. Figure 4.1 (left) illustrates the relative performance of FPGA execution compared to sequential CPU Java execution[1] when running the Discrete Fourier Transform (DFT) application. As shown in Figure 4.1 the DFT original implementation for the FPGA execution performs up to 17% slower (for small datasets) than CPUs.

After thoroughly inspecting the generated code, the main reason behind this performance degradation is that TornadoVM was tuned and optimised for CPU and GPU acceleration rather than FPGAs. Unlike CPUs and GPUs, FPGAs require hardware-specific annotations to be passed along with the generated OpenCL code to the underlying HLS tools. These annotations will hint the HLS to produce an optimal hardware design by

[1] See 4.4 for a detailed discussion on the experimental setup.

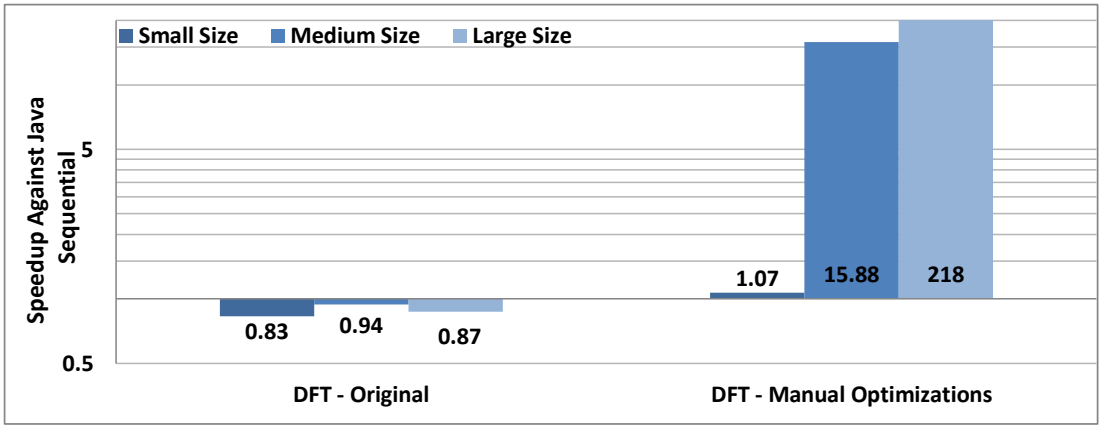


Figure 4.1: Initial results of TornadoVM generated OpenCL code a DFT application running on an FPGA: a) un-optimised (left), and b) with manual optimisations (right).

triggering optimisations, such as loop pipelining. To assess the impact of these annotations, the exploitation of the optimisation space through adding OpenCL pragmas (i.e., pragma unroll, thread attribute) to the auto-generated OpenCL kernel was performed. The performance results achieved through this activity are depicted in Figure 4.1 (right). As shown, the manually optimised DFT application outperformed the sequential vanilla Java code executed on the CPU by up to 218×. These results were in line with the well-documented *performance portability* [CC19] challenges of OpenCL across different hardware accelerators. Based on these findings, the TornadoVM compiler was augmented to automate the above mentioned process.

4.2 FPGA Acceleration of Managed Languages

To address the integration and performance portability challenges mentioned in Section 3.2.3, the original TornadoVM framework has been augmented to provide the following:

(a) Support for JIT compilation and emulation mode for seamless execution of Java applications on FPGAs

(b) A set of compiler extensions to enable optimisations which aim to replace the manual code interventions initially performed on the OpenCL generated kernels (Section 4.3).

(c) A complete tools chain that enables users to write a program "once" and "run it

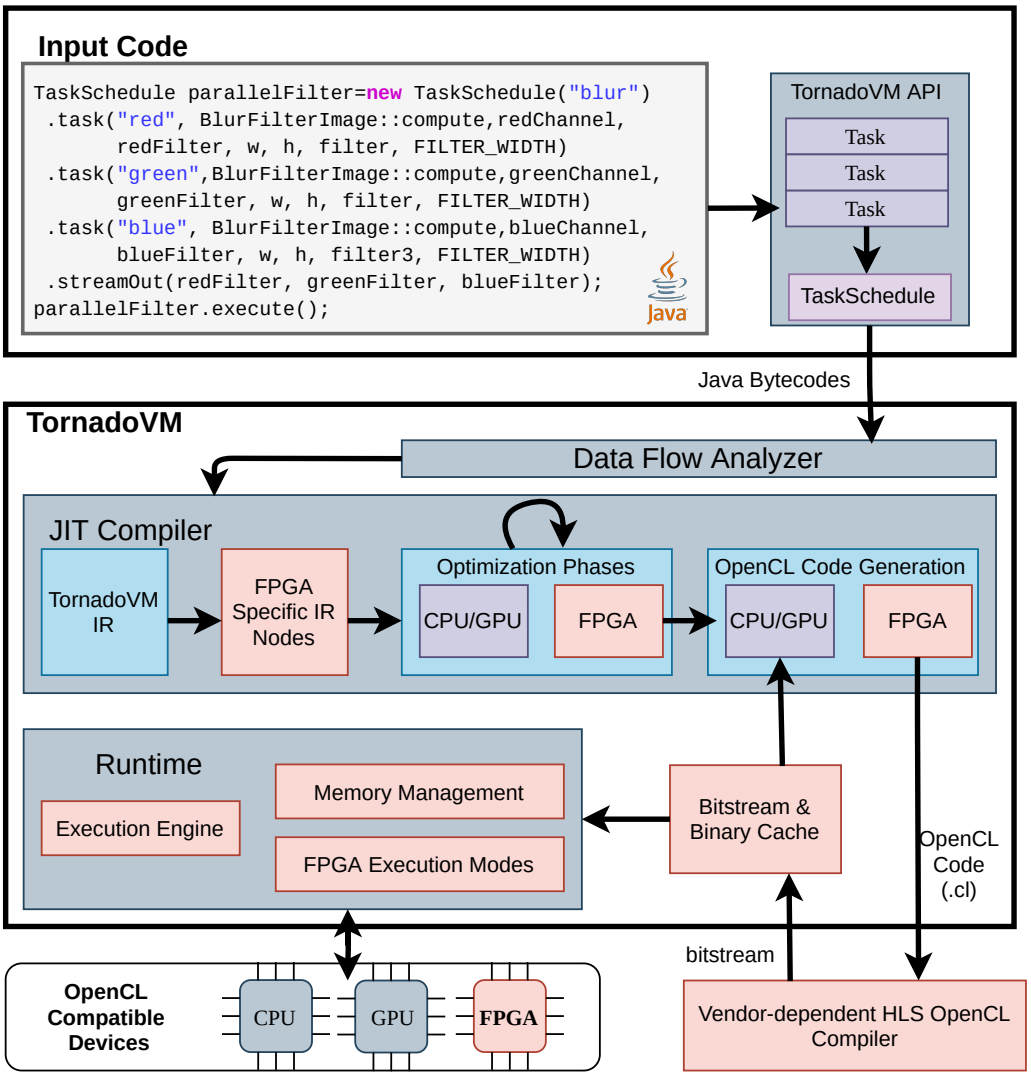


Figure 4.2: TornadoVM Overview: The existing components are illustrated with blue while the FPGA extensions are depicted in pink.

anywhere", even on FPGAs, while taking advantage of hardware acceleration to achieve better performance.

Figure 4.2 presents the extensions made to TornadoVM, showcasing how the new approach can be practical for harnessing the FPGA technology within the Java language. The existing components of TornadoVM are illustrated with blue, while the applied extensions are depicted in pink.

To generate FPGA-compatible code, the TornadoVM OpenCL backend has been extended instead of implementing a new backend for generating HDL similarly to other approaches [ABCR10, KPZ+16]. The extension of the current OpenCL backend to

support seamless FPGA execution has the following advantages: *a)* increasing industrial support and maturity of OpenCL compilers and hence constantly improved performance on FPGAs, *b)* plug-and-play of customised and proprietary bitstream kernels that follow OpenCL semantics in cases that there is no access to the source code (legacy or licensed code), and *c)* it is consistent with the rest of the TornadoVM framework, thereby increasing maintainability.

The remaining of this section describes the individual changes made to the TornadoVM compiler (Section 4.2.1), runtime (Section 4.2.2), and memory management (Section 4.2.3).

4.2.1 Extensions to the JIT Compiler

As shown in the work-flow presented in Figure 4.2, the input Java code is compiled to Java bytecodes using the standard Java compiler (`javac`). Then, the *TornadoVM Data Flow analyser* [CFP+18c] exploits the data dependencies and builds an initial Intermediate Representation (IR) graph of the input program. The generated IR graph is compiled down to the target architecture following the two-stage compilation approach illustrated in Figure 4.3. At the first stage, Java bytecodes are JIT-compiled to OpenCL C, while at the second stage, the OpenCL C code is compiled to FPGA bitstream by the external toolchains of the FPGA vendors.

During the first-stage compilation, the input IR graph is optimised and specialised through the TornadoVM JIT compiler before the final OpenCL C code emission. Since the TornadoVM JIT compiler is a superset of the Graal compiler [WWW+13, DSW+13], it inherits both its existing set of optimisations and its IR representation. Hence, it employs not only device-specific optimisations and specialisations (e.g., for GPUs, multicore CPUs) but also standard compiler optimisations (e.g., loop unrolling, global value numbering, common subexpression elimination, etc.) derived from the Graal compiler.

Integration with HLS Software: After completing the first-stage compilation, the generated OpenCL C code is automatically forwarded to the HLS compilers (e.g., Intel `aoc`); which subsequently performs the second-stage compilation from OpenCL C into the FPGA bitstream. Once the FPGA bitstream is generated, it is stored into the bitstream cache inside TornadoVM. This facilitates the reuse of the bitstreams based on the requirements of the Java programs. Although the current state of the toolchain integrates Intel FPGAs, this system has been designed and implemented in a modular

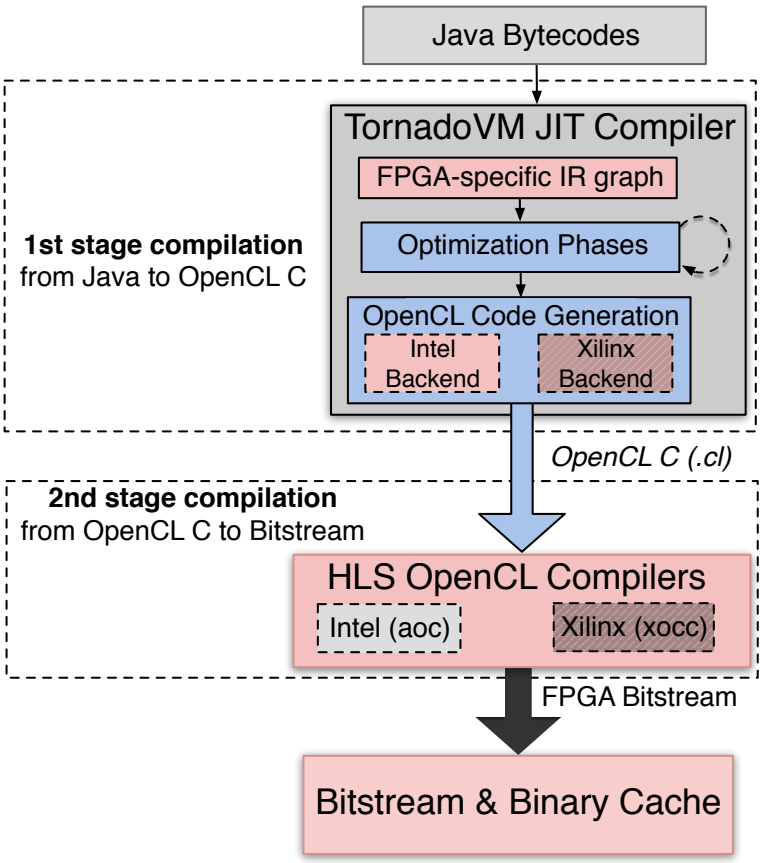


Figure 4.3: Two stage compilation: 1) from Java to OpenCL C, and 2) from OpenCL C to FPGA Bitstream.

way to support multiple HLS-specific backends. This was the springboard that allowed the easy expansion of the toolchain for hosting multiple state-of-the-art HLS compilers, such as Vivado [Cha16] HLS from Xilinx, and therefore target cloud-native solutions, such as the AWS cloud.

4.2.2 Runtime Extensions

The initial investigation presented in Section 4.1 illustrated how ahead-of-time FPGA compilation can be achieved through manual intervention. However, ahead-of-time FPGA compilation requires users to perform the HLS compilation stage manually. This section presents the various execution modes added to the runtime that allow programmers to adapt code execution based on their requirements. These execution modes aim to compensate for the long design cycle that is the aftermath of the ample time that the bitstream generation takes. In addition, Section 4.4 further analyses the

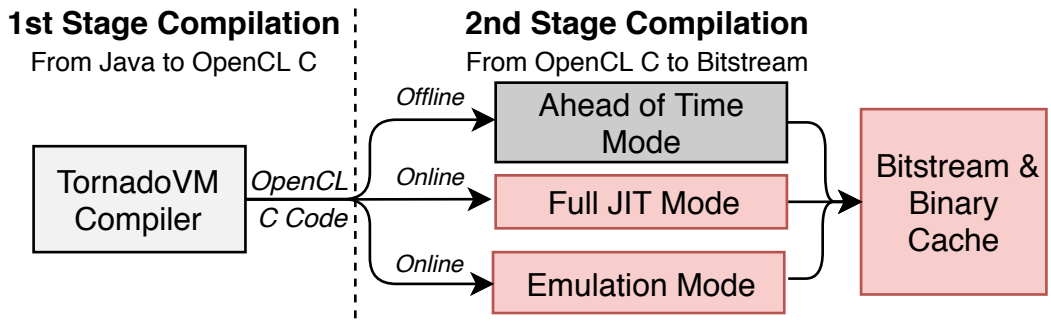


Figure 4.4: An overview of the execution modes. The extensions are illustrated in pink.

overheads of the compilation process.

The execution modes illustrated in Figure 4.4 are namely the following: the *Full JIT mode*, the *Ahead of Time Mode* and the *Emulation Mode*. The provision of these execution modes allows Java applications to be automatically adapted based on their requirements.

Ahead-of-Time mode: This mode alleviates the overhead of FPGA synthesis by allowing the plugin of precompiled bitstreams to TornadoVM during execution. The omission of the latency of the second compilation stage (from OpenCL C to bitstream) makes this mode suitable for applications that are sensitive to JIT compilation times (e.g., fast start-up applications or low energy requirements). In addition, since this mode allows users to plug in their bitstream implementations, disaggregated machines can be used for FPGA bitstream generation without any limitations, such as system resource utilisation or licensing issues.

Full JIT mode: This mode enables the end-to-end JIT compilation and execution of Java code onto FPGAs. This is achieved by creating a separate Java thread that makes direct calls to the HLS compilers for OpenCL (e.g., Intel `aoc` compiler for FPGAs). The HLS compilers for OpenCL follow the traditional process for compiling the OpenCL code into the FPGA bitstream [NSP+16]. Once the bitstream is generated, the runtime system stores it into the bitstream cache and marks the Java method *ready* to be executed on the FPGA. Finally, the runtime system loads the generated binary onto the FPGA device, creates the OpenCL context for a given program, and copies all data required to launch the kernel. The *Full JIT* compilation from the original Java source code to a fully functional hardware design is enabled through this mode. However, this JIT compilation process typically requires up to two hours (see Section 4.4) due to a timing consuming stage of the HLS compilation pipelined called placement and routing. Thus, the *Full JIT* mode is suitable for long running and server applications, in which the

compilation time is offset by the speedups achieved from FPGA acceleration.

Emulation mode: FPGA vendors provide tooling for FPGA emulation to reduce the overhead that is associated with the bitstream generation process. Through this emulation process the code intended for FPGA execution is compiled to the CPU. This compilation process generates optimization and resource utilization reports to identify potential bottlenecks before the bitstream generation takes place.

The emulation mode is used for fast prototyping, initial debugging, and functional validation of the generated FPGA kernels. This mode is not intended for any performance evaluation, as the emulated kernel code runs on a CPU thread and not on the physical FPGA device. This mode is added to aid developers at the initial development stage or debugging, since it avoids the HLS compilation overheads. Furthermore, it can provide an estimated view of the resource utilisation and any compiler warnings associated with the Java code. More importantly, widely available Java Integrated Development Environments (IDEs) (e.g., Eclipse, IntelliJ, NetBeans) can be used to develop software and test Java applications on FPGAs. Hence, developers with no HLS background can experiment by writing pure Java code using standard IDEs and assess whether their code can functionally run on an emulated FPGA. The use of standard IDEs in the development process is also applicable to the other two execution modes.

4.2.3 Memory Management

Figure 4.5 provides a high-level overview of how variables stored on the host side in the Java heap can be allocated to the device through JNI calls to OpenCL functions. The memory management between the host and the FPGA works as follows: the first time TornadoVM utilises an FPGA, it allocates a large amount of global memory on the device that acts as a managed heap (similarly to a Java heap). The rationale behind this managed on-device heap is to minimise the allocation times on the target device. TornadoVM performs only a single allocation while performing all data transfers between the host and the FPGA transparently to the user. This memory management system is adopted from the original framework. However, the OpenCL specification imposes a limiting factor regarding the maximum single buffer size. OpenCL does not allow single arrays larger than 1GB to be copied to a device at once [WPHZ17]. Furthermore, the proposed toolchain increases the bandwidth between the CPU and the FPGA by using page-locked (or pinned) memory. This enables OpenCL programs to use Direct Memory Accesses (DMA), thereby enhancing the performance of memory transfers. To use pinned memory on the FPGA, the TornadoVM runtime is required to

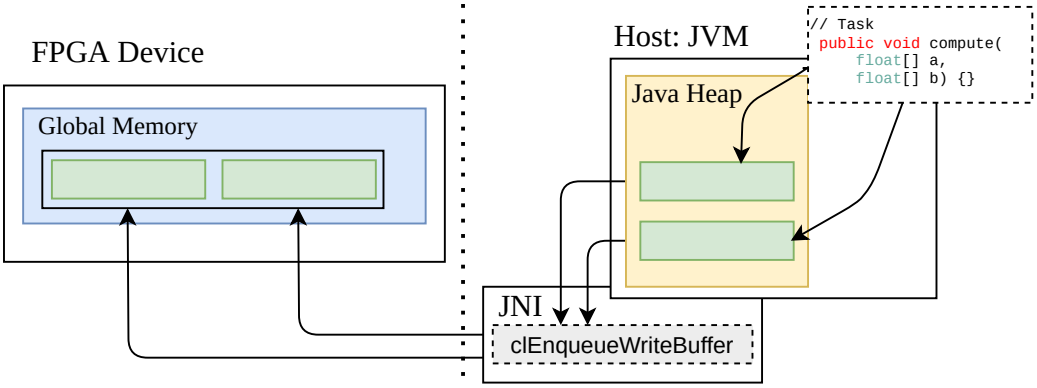


Figure 4.5: Abstract overview of the FPGA memory management scheme.

allocate memory using the OpenCL flag `CL_MEM_ALLOC_HOST_PTR`[2]. The Java stack-frames (memory regions that include the return address and addresses of all input/output buffers on the Java heap) and all Java arrays required for the kernel execution are copied to this allocated region.

The extended TornadoVM memory manager copies all data to the FPGA's global memory, and it keeps track of all host variables that have been copied to the FPGA. The toolchain classifies all arrays copied to the FPGA as read-only, write-only, or read-write during the runtime data analysis. Read-only Java arrays are persisted in the global memory of the device without copying them back to the host memory. On the contrary, write-only and read-write Java arrays are copied back to the host memory to make their updated values visible to the Java applications. Since TornadoVM dispatches and runs OpenCL code on the FPGA, all operations are, by default, non-blocking. This means that the operations regarding data transfers for the OpenCL kernel are non-blocking between the FPGA and the main host. Therefore, an extra barrier is added at the TornadoVM bytecode level to wait for the last kernel to be finished before performing the final copy from the device (FPGA) to the host to obtain the results.

A functional solution by enabling a two-stage compilation process does not guarantee high-performance without manual code-tunning, as it was highlighted in Section 4.1. Therefore, to run efficiently on FPGAs, there is a need to augment the compiler with FPGA awareness during compilation.

[2] `https://intel.ly/2J0mQFj`

Listing 4.1: Example of Java code snippet for the dft method.

```
1  void dft(float[] inreal,float[] inimag,float[] outreal,
2              float[] outimag, int[] inputSize) {
3  for (@Parallel int k = 0; k < n; k++) {
4      float sumreal = 0;
5      float sumimag = 0;
6      for (int t = 0; t < n; t++) {
7        float angle = ((2 * Math.PI() * t * k) / (float) n);
8         sumreal += (inreal[t] * (Math.cos(angle)) + inimag[t]
9         * (Math.sin(angle)));
10       sumimag += -(inreal[t] * (Math.sin(angle)) + inimag[t]
11        * (Math.cos(angle)));
12     }
13     outreal[k] = sumreal;
14     outimag[k] = sumimag;
15  }
16 }
```

4.3 Compiler Optimisations Targeting FPGAs

As discussed in Section 4.1, although the initial OpenCL-generated code was functionally correct, its performance was not portable on FPGAs. Therefore, the code must be specialised to exploit the features available on each hardware accelerator. In the case of FPGAs, this thesis proposes a set of compiler optimisations to automatically optimise Java programs for FPGAs without modifying the source code provided by the user.

4.3.1 Extensions to the JIT Compiler

To enable FPGA-specific optimisations, extensions to the IR of the JIT compiler with FPGA-related nodes were made. Briefly, the FPGA compilation flow works as follows: first, the TornadoVM runtime invokes the compiler to build the IR graph that represents the input Java method to be compiled. Consequently, the extended compiler specialises the IR graph for FPGAs by introducing new nodes and optimisation phases. After the code is optimised and specialised for FPGAs, the final OpenCL C code is generated (Figure 4.4, 1st stage compilation). Finally, the generated code is handled by the extended runtime system, which drives the 2nd stage compilation (Figure 4.4) based on the corresponding execution modes.

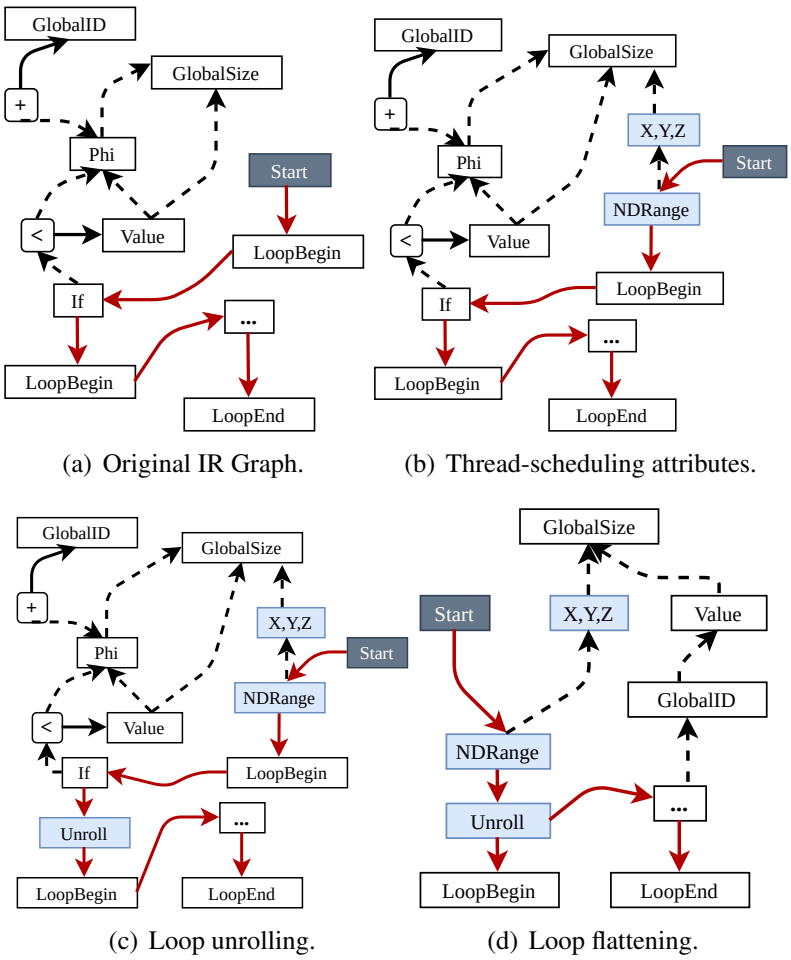
The introduced FPGA compiler optimisations are a) thread-scheduling attributes, b) loop unrolling, and c) loop flattening. Listing 4.1 shows a Java code snippet for the

Discrete Fourier Transform (DFT) method that will hereafter be used to describe the optimisations above. The listed code contains a method with two nested `for` loops (lines 3 and 6) with the computation residing inside the nested loop. Note that the first loop is annotated with the `@Parallel` annotation, previously discussed in Section 2.3.1. Figure 4.6 illustrates how this work extended the original TornadoVM compiler. As shown, several compiler transformations are automatically applied to the IR of the code in Listing 4.1.

Figure 4.6(a) shows the IR graph that corresponds to the code after initially invoking the TornadoVM compiler. As shown, there are two groups of nodes: data-flow nodes connected by black dashed arrows and control-flow nodes connected by red arrows. Furthermore, each method begins with the `Start` node. The graph in Figure 4.6(a) shows two `LoopBegin` nodes that correspond to the two loops from the input Java code. To compute the bounds for each loop, a `phi` node along with an `if` condition is added in the IR. Since the compiler extensions are implemented in TornadoVM, they reuse two new nodes for computing the corresponding indices in OpenCL; namely `GlobalID` and `GlobalSize`.

Attributes for Thread-Scheduling: Initially, for kernels targeting OpenCL compatible GPUs, the IR information containing the global indices was sufficient for thread indexing. However, by keeping only the default OpenCL global indices, the generated kernel was designed for single-threaded FPGA execution. Consequently, when launching multiple threads on the FPGA, they pointed to wrong memory locations and thus generated erroneous results. To address this problem, the IR is extended with a new node to generate OpenCL C attributes before the function header of the kernel. These attributes specify the thread selection (number of threads per block for each dimension — 1D, 2D, or 3D) with which the kernel should be executed. This compiler optimisation is presented in Figure 4.6(b). A new node called `NDRange` is inserted right after the `Start` node. This node points to an additional data-flow node that indicates the values for the thread-blocks in 1D, 2D, and 3D (x, y, z), respectively. These values depend on the global size of the compute kernel.

Loop Unrolling: The second FPGA optimisation applied is explicit loop-unrolling, a widely used optimisation for improving OpenCL performance on FPGAs [WHU17]. Initial investigation of the optimisation space showcased that improved performance can be achieved through pragmas rather than apply loop unrolling before generating the OpenCL kernels. The inner loop analysis is performed by inspecting the loop bounds. The inspection aim to detect bounds assigned with constants values instead



(a) Original IR Graph. (b) Thread-scheduling attributes.

(c) Loop unrolling. (d) Loop flattening.

Figure 4.6: IR compiler transformations that are automatically performed by the implemented extensions to the JIT compiler.

of being dynamically assigned. For cases where the input code can contain several nested loops, the compiler always tries to perform loop unrolling to the innermost loop. For example, assuming that a loop is a candidate to be unrolled, the newly introduced phases in the TornadoVM compiler insert a new control-flow node (i.e., `Unroll`) in the IR before the `LoopBegin` node of the corresponding candidate loop for unrolling. Figure 4.6(c) highlights the optimisation in which the inner unrolled loop is annotated with the `Unroll` node. The loop unrolling phase uses as a basis the default unroll phase of Graal, which means it considers only loops with up to 128 dependency-free iterations as *unrollable*. Consequently, the OpenCL code generator reads the new `Unroll` node. This node triggers the emission a `pragma unroll` in the OpenCL C code, leaving the underneath HLS compiler to decide the unrolling factor.

Loop Flattening: The final compiler optimisation that is applied is loop-flattening.

The original compiler performs node replacement to substitute `for` loops with the OpenCL indexing primitives (e.g., `get_global_id`). However, it maintains the `for` loops in the OpenCL C code if the kernel processes more elements than the available threads on the target device. Since the extensions made to the TornadoVM compiler specialise the IR for FPGA execution, they also specialise the input index space. Therefore, loop nodes can be safely replaced by the OpenCL indexing primitives. It is important to note that the compiler extensions only flatten the parallelised loops by replacing the loops with the OpenCL indexing primitives. If a loop is computed sequentially, the compiler will preserve the loop nodes. The loop flattening optimisation is highlighted in Figure 4.6(d), in which the outermost loop is removed along with every data dependency associated with it.

4.3.2 Generated FPGA-Optimised OpenCL C code

Figure 4.7 provides a sketch of the generated OpenCL code for FPGAs reflecting all described optimisations for the input Java code of Listing 4.1. This Java code contains the user-defined annotation (`@Parallel`) that indicates that a loop in the program can be transformed for parallel execution. The left side of Figure 4.7 shows the generated OpenCL code without automatically applying the implemented compiler optimisations. In contrast, the right side shows the generated code highlighting the outcomes of the compiler optimisations. The yellow block on the right side highlights the attribute for determining the number of work-items (threads) used for thread scheduling on the FPGA. In this case, it is set to 64 elements, as this number has been shown to offer maximum performance on Intel FPGAs [SEEZ19, WOL+17]. The red block highlights the outer loop which is targeted to be flattened. On the right-hand side the green block showcases how during its post-optimisation phase the loop is only indexed by the *get_global_id* OpenCL intrinsic. This optimisation simplifies the generated hardware circuits on the FPGA and, therefore, increases performance. Finally, the blue block highlights the loop unrolling for the FPGA with a factor of two through the *pragma unroll* OpenCL Intel FPGA directive before the innermost loop.

Once the OpenCL FPGA code is generated, a call to the underlying OpenCL compiler (e.g., the Intel `aoc` compiler) is required to compile the OpenCL C source code to the FPGA bitstream as explained in Section 4.2.1.

```
__kernel void compute(__global uchar *_heap_base,
                      ulong _frame_base, ... ) {
// variable declaration
...
__global ulong *_frame=(__global ulong*)
&_heap_base[frame_base];

base0 =  (ulong) _frame[6];
base1 =  (ulong) _frame[7];
base2 =  (ulong) _frame[7];
tid   =  get_global_id(0);
...
i8  =  *((__global int *) &_heap_base[base0]);
for(;tid < maxElements) {
  ...
  f10 = 0.0F;
  i11 = 0;
  for(;i11 < i8;) {
      ...
  }
  ul_38 = base1 + index;
  *((__global float *) &_heap_base[ul_38]) =
  ul_37 = base2 + index;
  *((__global float *) &_heap_base[ul_39]) =
  i_40 = get_global_size(0);
  i_41 = i_40 + tid;
  tid = i_41;
}
```

```
// Scheduling attributes
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void compute(__global uchar *_heap_base,
                      ulong _frame_base, ... ) {
// variable declaration
...
__global ulong *_frame = (__global ulong *)
&_heap_base[_frame_base];

base0 =  (ulong) _frame[6];
base1 =  (ulong) _frame[7];
base2 =  (ulong) _frame[7];
tid   =  get_global_id(0);            // Loop flattening
...
i8  =  *((__global int *) &_heap_base[base0]);
...
f10  =  0.0F;
i11  =  0;
#pragma unroll 2     // Loop unrolling with factor 2
for(;i11 < i8;) {
   ...
}
ul_38  =  base1 + index;
*((__global float *) &_heap_base[ul_38])  =  result1;
ul_37  =  base2 + index;
*((__global float *) &_heap_base[ul_39])  =  result2;
}
```

Figure 4.7: Sketch of the generated OpenCL code specialised for FPGAs (LHS: Original TornadoVM generated for GPUs and RHS: TornadoVM code generated for FPGAs).

4.4 Evaluation

This section presents an analysis of the various performance aspects of the complete toolchain. Section 4.4.1 outlines the experimental methodology, benchmarks and hardware configuration used. Section 4.4.2 presents the attained performance improvements against various baselines. Section 4.4.2.1 and 4.4.2.2 presents the runtime overhead analysis and effect of the compiler optimisations respectively. In addition, Section 4.4.3 outlines the HLS compilation overheads and Section 4.4.4 presents the resource utilisation on the FPGA device.

4.4.1 Experimental Setup and Methodology

The performance of the FPGA executed code for the end-to-end toolchain is evaluated against the *peak* performance of single and multithreaded Java implementations compiled with the server compiler (C2) of OpenJDK [PVC01b]. Also, to guarantee that the JVM has been warmed up, up to 150 iterations per benchmark were performed and the mean of the consecutive ten runs is reported. To ensure the functional correctness of the generated FPGA code (Section 4.2.2), the validation has been performed through execution in emulation mode. After that, all FPGA bitstreams were generated while using the full JIT mode for each benchmark.

Table 4.1: Input and data sizes for the given set of benchmarks. Input size corresponds to the number of parallel iterations while the data sizes correspond to the in/out data transfers.

Benchmark	**Input size**			**Data-In (MB)**	**Data-Out (MB)**
	small	**medium**	**large**	**max**	**max**
VectorAdd	32768	1048576	67108864	540	268
Grayscale	256	4096	32768	540	268
BlackScholes	256	65536	33554432	268	536
RenderTrack	64	1024	8192	268	200
N-Body	256	8192	32768	6	3
DFT	64	65536	262144	2	1

The execution time of the FPGA code is also reported using the arithmetic mean of ten consecutive runs, similarly to the CPU-executed code. Furthermore, all reported numbers correspond to end-to-end executions, which include the times (i) for loading the bitstreams into the FPGA, (ii) executing the kernels, (iii) copying the data from the host to the FPGA memory, and (iv) copying back the data from the FPGA memory to the host (CPU). Finally, each benchmark has been evaluated against three different workloads –small, medium, and large– with data sizes (Table 4.1) increasing by orders of magnitude, varying from 1MB to 540MB. The large size corresponds to the maximum size permitted by the HLS compiler for mapping each generated circuit on the available FPGA device.

4.4.1.1 Benchmarks

Regarding the performance evaluation, two standard applications (`Vector-Add` and `BlackScholes`), two variations of computational dwarfs [KFAB16] (`NBody` and `DFT`), and two computationally intensive kernels for image processing (`RenderTrack` and `Grayscale`) were used. For all benchmarks, both sequential and multithreaded Java implementations have been ported, verified and evaluated.

4.4.1.2 Experimental Setup

Table 4.2 outlines the experimental hardware platform to evaluate and validate the performance gains of the proposed toolchain. The accelerator card is an Intel Arria 10 FPGA (*10AX115N3F40E2SG*). Also, the Arria 10 FPGA offers native `IEEE 754` single-precision floating-point operations through its DSP blocks [Intc]. The FPGA

Table 4.2: Experimental Platform for FPGA Experimentation.

Hardware	
Processor	Intel Core i7-7700 @ 4.2GHz
Cores	4 (8 HyperThreads)
RAM	64GB
IGPU	Intel HD Graphics 630
FPGA	Nallatech 385A, Intel Arria 10 FPGA, Two banks of 4GB DDR3 SDRAM each
Software	
OS	CentOS 7.4 (Linux Kernel 3.10.0-693)
OpenCL (CPU)	2.0 (Intel)
OpenCL (GPU)	1.2 19.43.14583 Intel OpenCL
OpenCL (FPGA)	1.0 (Intel), Intel FPGA SDK 17.1, HPC Board Support Package (BSP) by Nallatech
JVM	Java SE 1.8.0_131 64-Bit JVMCI VM [3]
Java Heap	16GB

frequency for all the kernels is automatically determined by the Intel OpenCL compiler and ranges from 176 to 218 MHz.

4.4.2 Performance Analysis

To conduct the performance evaluation three different configurations against various baselines have been used. The first two concern the performance improvements over sequential and multithreaded Java execution, while the last one assess the acceleration performance over an Intel Integrated GPU.

FPGA versus sequential Java code: Figure 4.8 shows the performance of the FPGA executed code against the sequential Java code. As shown, FPGA execution for small workloads exhibits performance slowdowns across all benchmarks except for `DFT`. This is due to the time spent in data transfers which is significantly higher than the FPGA computation time. In the case of memory-bound benchmarks, such as `VectorAdd`, the performance slowdown can reach up to 0.002× . Regarding `Grayscale`, `BlackScholes` and `RenderTrack`, although they perform worse compared to sequential Java for small workloads. On the contrary, they show performance scalability while increasing data sizes, with peak speedups of 11×, 15× and 30×. Moreover, `NBody` shows a similar

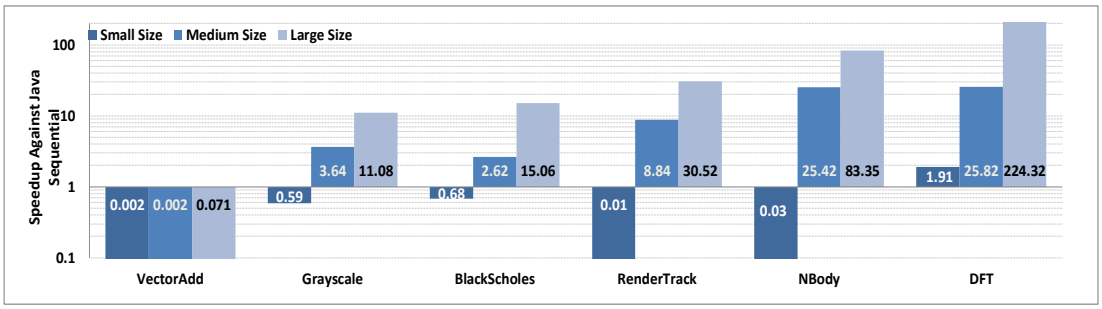


Figure 4.8: Speedup of Intel Arria 10 FPGA against sequential Java for small, medium and large data sizes.

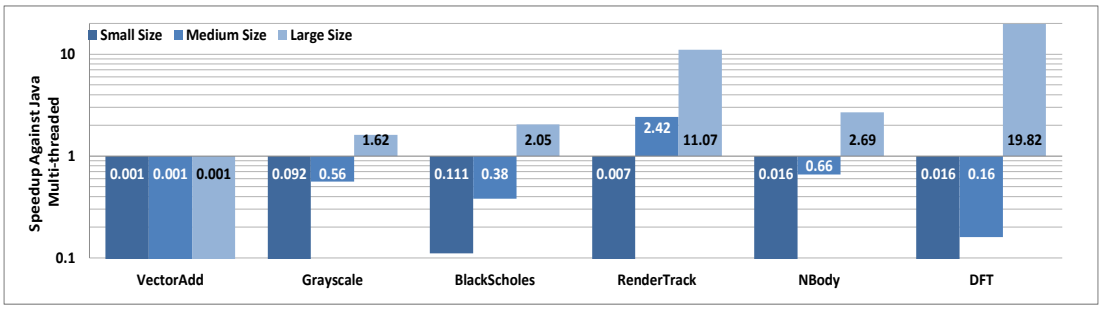


Figure 4.9: Speedup of Intel Arria 10 FPGA against multithreaded Java (8 threads) for small, medium and large data sizes.

performance trend with a peak speedup of 83×. Finally, for highly computational benchmarks, such as `DFT`, the FPGA outperforms the CPU-executed sequential Java code for all input sizes by up to 224× .

FPGA versus multithreaded Java code: Figure 4.9 shows the performance of the FPGA executed code against the multithreaded Java code. All benchmarks utilise the maximum number of available threads in the system (eight), except `RenderTrack` for which *Hyper-Threading* was deactivated since it was resulting in performance degradation. As shown in Figure 4.9, for large data sizes, FPGA execution outperforms the multithreaded Java implementations from 1.62× up to 19.82×. However, for small and medium input data sizes, the multithreaded Java code outperforms the FPGA executed code except for the `RenderTrack` benchmark. This a performance pattern is associated with the overhead of copying data to/from the FPGA.

FPGA versus an Intel HD Graphics 630 GPU: Figure 4.10 shows the performance of the FPGA executed code against TornadoVM running on an Intel HD Graphics 630 integrated GPU. The configuration of the `local- work-group` size

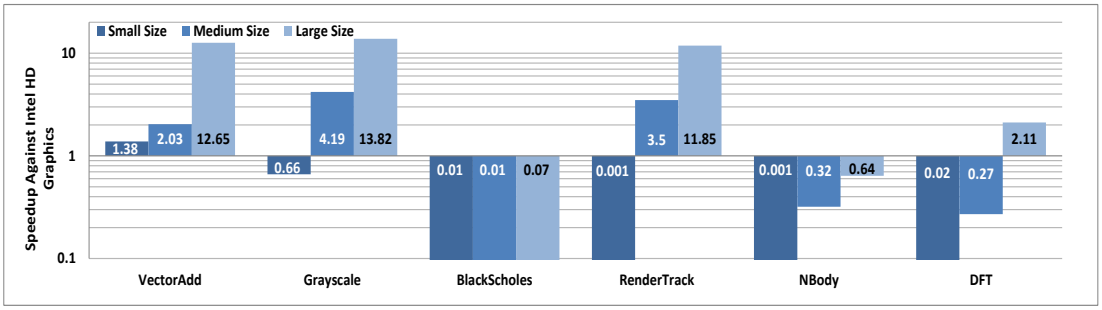


Figure 4.10: Speedup of Intel Arria 10 FPGA against Intel HD Graphics 630 for small, medium and large data sizes.

is not manually tuned, instead the `local_ work_size` attribute is left empty for the `clEnqueueNDRangeKernel`. Therefore, the Intel driver and its OpenCL implementation will automatically determine how to distribute the global work-items. As shown in Figure 4.10, for all benchmarks except NBody and Black-Scholes for large data sizes, FPGA execution outperforms the Intel Graphics up to 13.82×. The performance for medium workloads performance varies depending on the workload. For instance, workloads that make use specialised math operations exhibit better performance over the integrated GPU. However, the FPGA always performs worse for small sizes compared to the Intel HD Graphics card due to the overheads for copying the data to/from the device. This behaviour is expected, and for that reason, the FPGAs are recommended to be targeted in cases of computationally intensive workloads.

4.4.2.1 Runtime Overhead Analysis

To further understand the overall performance of the system, a break-down analysis of the end-to-end execution times of all benchmarks is presented in Figure 4.11. The analysis of the execution times is conducted only for the largest input data sizes. This decision was made as this data size highlights the impact of the data transfers between the host and the device memories. Each bar has four parts corresponding to: a) kernel execution time on the FPGA (*Kernel*), b) data transfers from host to device (*H2D*), c) data transfers from device to host (*D2H*), and d) the (*Rest*). The *Rest* includes the time for loading the binary on the FPGA and initialising low-level OpenCL data structures, such as the OpenCL context of each kernel.

Figure 4.11 shows that up to 18% of time is spent in transferring data from the host to the FPGA device (*H2D*) and backwards (*D2H*). In particular, `VectorAdd`, `BlackScholes` and `Grayscale` spent up to 10%, and `RenderTrack` up to 18% of their

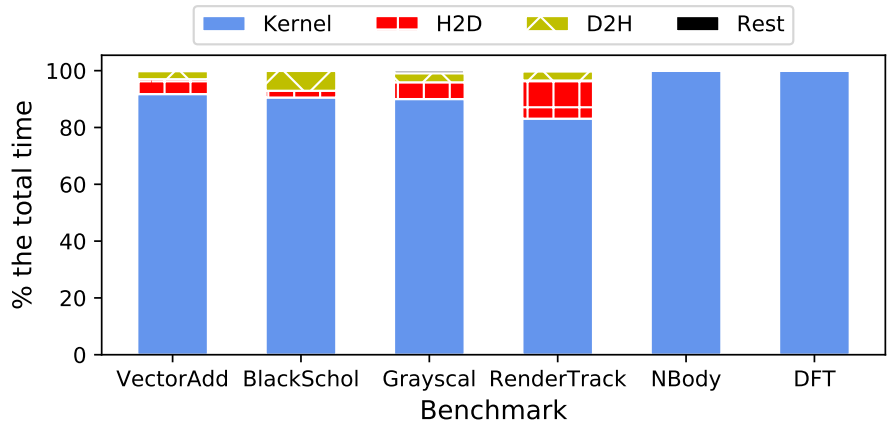


Figure 4.11: Breakdown of the total execution time of each benchmark.

total time in data transfers. On the contrary, the *Kernel* execution time is up to 99.9% for both computationally intensive benchmarks; `NBody` and `DFT`.

The `VectorAdd` benchmark is a particular case because it exhibits slowdowns, as illustrated in Figure 4.8, even though the kernel execution percentage is large enough to anticipate performance improvements. The reason is that this benchmark is memory intensive, and the current version of the toolchain does not support memory-specific optimisations, such as local or constant memory (Section 5). Hence, there is significant memory traffic in the global memory (DRAM). However, in TornadoVM all data is stored in a single buffer which does not support anti-aliasing and coaleched accesses. Therefore, when a task has minimal compute operations while operates heavily on read/write from global memory, the performance can be penalised significantly. One potential optimisation would be to automatically vectorise load and store operations with OpenCL by specialising the kernel. Finally, the time for loading the binary and initialising OpenCL contexts (*Rest*) across all benchmarks is negligible.

4.4.2.2 Optimisation Phases Breakdown

The data obtained from exploiting the optimisation space, and reported in Table 4.3 shows the contribution of each phase to the overall *performance* shown in Figure 4.8.

The first phase includes only the *Thread Scheduling (TS)* optimisation and shows two different thread-block configurations; one with 32 threads and one 64 threads. These numbers suggested by the available technical report on the Intel FPGA. Note that during experimentation any thread configuration of more than 128 caused errors on the HLS software side. In this work, the main observation is that the thread-block that offers the best performance for the specific FPGA board is the 64 thread-block in

Table 4.3: The impact of each optimisation phase in performance. The first optimisation phase includes Thread-Scheduling (TS), the second phase applies Loop Unrolling (LU) along with scheduling with 64 threads (TS_64). The final phase includes all previous optimisations and Loop Flattening (LF).

Benchmark	**Thread Scheduling (TS)**		**TS_64 + LU**	**TS_64 + LU + LF**
	TS_32	**TS_64**		
VectorAdd	0.00002×	0.0001×	N/A	0.071×
Grayscale	9.42×	10.01×	N/A	11.08×
Black-Scholes	15.85×	15.97×	N/A	15.06×
RenderTrack	21.23×	28.16×	N/A	30.52×
NBody	11.04×	31.20×	66.75×	83.35×
DFT	35.44×	56.31×	214.64×	224.32×

OpenCL. This configuration is achieved a maximum performance improvement of up to 56× over the Java sequential code compared to 35.44× for the 32 thread configuration. Besides, as the Thread Scheduling column of Table 4.3 shows, all benchmarks showcase the best performance at the 64 thread configuration. Thus, 64 threads were used in the experiments shown in Figures 4.8, 4.9 and 4.10.

The second phase applies the loop unrolling (LU) optimisation on top of the first phase. Note that this optimisation is only applicable for applications in which an inner unrollable loop exists. Therefore, N/A reflects to non-applicable applications. In this benchmark selection only `DFT` and `NBody` are applicable. For `DFT` applying this optimisation phase (TS_64 + LU), the overall performance can be improved up to 214× (3.8× improvement). In addition, `NBody` reaches a performance improvement up to 66.75× (2.13× improvement).

The final phase is applicable to all benchmarks regardless of the structure of the kernel. This phase combines the previous optimisations with loop flattening (TS 64 + LU + LF). The maximum performance is achieved in the `DFT` with the overall performance to reach up to 224× compared to sequential Java. Moreover, the performance of the generated FPGA code is improved by 1.24× compared to the previous optimisation state of `NBody`. It can be noted that only `BlackScholes` showcases performance degradation due to this optimisation.

Optimising the unrolling factor mentioned above, as well as the work-group configurations can improve the overall performance of the generated kernel. However, this suggests optimisation iterations per use case or the extensive use for a performance modelling approach. Chapter 5 investigates alternative approaches to improve the efficiency of the generated code.

4.4.3 HLS Compilation & Binary Loading

Table 4.4 shows the HLS compilation times, binary loading times, and the sizes of the generated FPGA bitstreams for each benchmark. The HLS compilation time regards the 2nd stage compilation (Section 4.2.2), and it is the time spent for compiling the specialised generated OpenCL C code to a bitstream. The binary loading time is the time required to load the bitstreams and initialise low-level OpenCL primitives on the FPGA on behalf of the running program. As shown, while the binary loading time is in the range of milliseconds, the HLS compilation time can take up to 114 minutes (about 2 hours) to complete. Furthermore, the HLS compilation time includes the time for placement and routing, which is strongly related to the vendor tools and the complexity of the generated kernels. For instance, the `NBody` kernel reports the longest compilation time as it includes the loop unrolling optimisation, which utilises more private memory on the FPGA, and thus higher BRAM resources (Table 4.5). The increased latency in the HLS compilation times was the motivation for providing a set of execution modes in TornadoVM that can either perform a whole compilation for FPGAs at runtime (Full JIT) or load the bitstream of precompiled kernels (AOT) (discussed in Section 4.2.2). Nevertheless, OpenCL drivers for Xilinx and Intel are evolving quickly, thereby reducing compilation times and making JIT compilation more affordable [Int19]. Finally, Table 4.4 illustrated that both binary loading times and bitstream sizes are in the same range for all the benchmarks.

Table 4.4: Bitstream size, time for loading onto the FPGA, and HLS bitstream generation time.

Benchmark	**Bitstream Size (MB)**	**Load Bitstream (ms)**	**HLS Compilation (minutes)**
VectorAdd	172	22	48
Grayscale	173	23	52
BlackScholes	174	24	54
RenderTrack	173	23	44
NBody	173	24	114
DFT	173	22	68

Table 4.5: Resource utilisation as reported by the Intel High Level Synthesis Compiler (AOC).

Benchmark	**LUTs**	**FFs**	**DSPs**	**BRAM**
VectorAdd	145535 (19.5%)	275521 (18.4%)	72 (3%)	570 (39.4%)
Grayscale	117928 (15.7%)	230040 (15.4%)	72 (3%)	494 (34.0%)
Black-Scholes	186348 (24.9%)	306361(20.5%)	490 (20.7%)	935 (64.7%)
RenderTrack	118582 (15.9%)	238742 (16%)	72 (3%)	514 (35.5%)
NBody	174036 (23.3%)	329764 (22.1%)	120 (5.1%)	1291 (89.3%)
DFT	146418 (19.6%)	264652 (17.7%)	109 (4.6%)	748 (51.7%)
Resources	**747080**	**1494160**	**2367**	**1446**

4.4.4 Resource Utilisation

Table 4.5 shows the FPGA resource utilisation of four different hardware components – Look Up Tables (LUTs), Flip Flops (FFs), Digital Signal Processing (DSPs), and Memory Blocks (BRAM) – for each benchmark. As shown, the utilisation of the LUTs varies between 15.9% and 24.9% of the total capacity of the FPGA. In particular, `BlackScholes` utilises more LUTs and DSPs than the rest of the benchmarks, as the generated OpenCL C code of `BlackScholes` contains 216 lines of code with complex control flow. The utilisation of DSPs is between 3 and 5.1% for all benchmarks, except `BlackScholes` which is at 20.7% due to its code complexity. Regarding the BRAM utilisation, `NBody` is a particular case occupying up to 89.3% of the available resources BRAM resources. This high utilization is a result of the extensions to the TornadoVM JIT compiler that unroll two of the innermost loops.

Overall, these results indicate that the current set of benchmarks utilises roughly one-fourth of the FPGA available resources, except BRAMs. BRAMs show higher utilisation due to loop unrolling, which duplicates memory access and the intermediate stored values. Finally, the low utilisation of FPGA resources could provide enough space to accommodate multiple compute kernels per hardware design, resulting in higher performance.

4.4.5 Discussion

By carefully analysing the benchmarks and the obtained results of FPGA execution, a set of technical guidelines to can be obtained. These guidelines aim to highlight when FPGA acceleration is suitable for application written in Java.

Applications not suitable for FPGAs: `VectorAdd` is a case in which during runtime there is a need to copy a significant amount of data. However, this data only aims to compute just a few operations, thus this is not suitable for FPGA execution. This is attributed to modern CPUs, which operate at a much higher frequency than FPGAs (GHz versus MHz), can perform a larger number of such operations in less time. Also, the Java Hotspot compiler [AAZ+18, Ora14, PVC01b] can utilise high-performance special vector instructions and operations (e.g., fused multiply-add (FMA)).

Applications suitable for FPGAs: Applications, such as `BlackScholes`, `Grayscale`, `NBody`, `DFT` and `RenderTrack` exhibit significant speedups when operating over large data sizes. These applications use the specialised hardware on the FPGA to accelerate the abundance of math operations they contain (e.g., sine and cosine). This is because FPGAs can perform these operations in few clock cycles. This class of applications exhibits speedups for all inputs corresponding to large input sizes.

Overall, based on the above findings, Java applications can merit FPGA execution. In more detail, libraries such as Deep Netts [SOV+20], in which heavy and repeated mathematical computations occur can be suitable for acceleration. Therefore, calls to specific libraries can be diverted for FPGA co-execution.

4.5 Summary

This chapter outlined and discusses a novel and practical approach that allows managed languages to achieve seamless and efficient FPGA code execution. This work extends beyond the prior work discussed in Section 3.2 on high-level language execution for FPGAs and the gap outlined in Section 3.2.3 for Java execution on FPGAs. The key difference is that it allows automatically and transparently the generation of specialised FPGA code without explicitly requiring any hardware-related annotations in the source code. Section 4.2.2 presented the required modifications to enable seamless integration with HLS software, as well as memory management features for catering to FPGA execution particularities. Moreover, Section 4.3 illustrated how a minimal set of compiler optimisations enabled performant code generation targeting FPGAs. The proposed toolchain is designed to adapt to the requirements of a software developer. Therefore, it is compliant with high-level Java IDEs and debuggers while offering three execution modes based on the given requirements. Finally, Section 4.4 presented the performance evaluation of the complete toolchain. For this purpose, several Java

benchmarks using the original TornadoVM API were tested.

The toolchain, which used an Intel Arria 10 FPGA, showcased speedups up to 19.8× and 224× over multithreaded and sequential Java code, respectively. Besides, it also achieved speedups up to 13.82× over parallel execution on an integrated GPU (i.e., Intel HD Graphics 630) using TornadoVM.

While FPGAs can yield high-performance, HLS compilation times can impose a key challenge for their integration with commodity computing systems. The next chapter investigates how a heterogeneous managed runtime in combination with a state-of-the-art JIT compiler, can exploit the memory hierarchy of GPUs seamlessly. It outlines several techniques to exploit the memory hierarchy of GPUs through explicit memory allocation on `private` or `local` memory or loop transformations, such as tiling and partial unrolling.

Chapter 5

Exploiting the Memory Hierarchy of GPUs via JIT Compilation

The previous chapter presented a novel mechanism for enabling seamless integration of FPGAs into managed runtimes. However, since FPGAs do not have a specific instruction set architecture (ISA), code generation heavily depends on the manufacturer's HLS, thus compilation times can be prolonged. Also, other common optimisations, such as local memory allocation on FPGAs are not only user-managed, but also an explicit cache hierarchy does not exist [Zoh18, HHA+18]. As a result, the benefits of JIT compilation techniques at run-time can be limited as they need to consider hardware and HLS particularities. On the contrary, generating code for other heterogeneous devices with pre-defined ISAs, such as GPUs, can showcase performance improvements, while using a combination of JIT compilation techniques.

Most of the programming languages used for programming GPUs (e.g., OpenCL, CUDA, OpenACC) expose to their APIs language constructs. Hence, developers must explicitly use them to optimise and tune their applications to harness the underlying hardware. This trade-off between GPU programmability and performance has been an active research topic. Section 3.3 discussed in detail work that revolves around polyhedral models [DYS+12, VCJC+13] or enhanced compilers for domain-specific languages (DSLs), such as Lift [SRD17] and Halide [RKBA+13]. These approaches either have high compilation overheads [BPCB10], which makes them unsuitable for dynamically compiled languages, or they still require developer's intervention to exploit the memory hierarchy of GPUs (through explicit constructs or annotations available in the parallel programming languages).

Optimising code written in a high-level languages for low-level GPU execution is a

challenging problem. However, JVM-based aggressive compilers have been addressing similar challenges for two decades. For instance, many aggressive optimisation techniques applied to Java, like escape analysis [Bla03], partial escape analysis, and scalar replacement work towards minimising the memory footprint, and thus Garbage Collection (GC) overheads. Moreover, being able to express low-level optimisations from a high-level programming abstraction is part of this process. JikesRVM [AAC+99] supported intrinsic functions, while Frampton *et al.* [FBC+09] extended these concepts by proposing metacircular intrinsics. Both of the above provide an experimented programmer with a utility to express architecture-specific optimisations. In addition, JIT compilers, such as the Hotspot C2 compiler [PVC01a], Mozilla IonMonkey [Moza], SpiderMonkey [Mozb] and Google TurboFan [Gooa] for v8 [Goob] use similar techniques. Finally, modern aggressive compilers, like Graal [DWS+13], offer a state-of-art infrastructure, like Snippets [SWU+15], to express low-level optimisations in Java.

This chapter presents an alternative approach for automatically exploiting the memory hierarchy of GPUs completely transparently to the users. The main approach is based on Just-In-Time (JIT) compilation and abstracts away low-level architectural intricacies from the user programs, while making its application suitable for dynamically compiled languages. The compiler extensions are implemented in the form of enhancements to the Intermediate Representation (IR) and numerous associated optimisation phases that can automatically exploit local memory allocations and data locality on GPUs.

The compiler optimisations for exploiting and optimising local memory have been evaluated against a set of reduction and matrix operations across three different GPU architectures. To showcase the performance benefits of this technique, two different baseline implementations were used; (i) the original code generated by TornadoVM that does not exploit GPU local memory, and (ii) the hand-written optimised OpenCL code.

Briefly, this chapter presents the following contributions:

- A novel JIT compilation approach for automatically exploiting local memory of GPUs without requiring manual intervention.
- An analysis on how compiler snippets can be used to express local memory optimisations by introducing *compositional compiler intrinsics*, that can be parameterised and reused for different compiler optimisations.
- A detailed evaluation across a variety of GPU architectures (i.e., AMD, Intel, and Nvidia GPUs), against the functionally equivalent auto-generated unoptimised

and the hand-written OpenCL code with the same optimisations. The experimental results showcase performance speedup of up to 2.5x versus the original code generated by TornadoVM, while reaching up to 97% of the performance of the manually optimised code.

The rest of this chapter is organised as follows: Section 5.1 outlines the JIT compilation process for exploiting local memory for reduce and matrix operations. Section 5.3 presents how Graal Snippets combined with runtime information regarding the available memory on the GPU introduce the Compositional Compiler Intrinsic (CCI). Section 5.4 explains in detail how the optimisation process is used for reduction and matrix operations. Section 5.5 showcases the performance benefits that the above process can yield without modifying the user code. Finally, Section 5.6 summarises the findings of this chapter.

5.1 Motivation: Tier-Memory for Locality in GPUs

Previously, Section 2.1.1.2 presented the characteristics of GPUs and Section 2.1.2.1 outlined the OpenCL memory hierarchy in Figure 2.3. However, efficient data locality on GPUs is crucial to achieve high-performance and avoid performance bottlenecks, such as memory aliasing [BAM13]. Section 3.3 outlined several state-of-art techniques and approaches that attempt to mitigate the complexity of data locality in GPUs with optimising compilers.

Exploiting the GPU Tier-Memory: Listing 5.1 outlines an OpenCL kernel for computing a Matrix Multiplication workload. This kernel utilises the global, local, and private memories of the GPUs. Additionally, it also showcases how to increase the efficacy of data locality by applying a common loop transformation, such as loop tiling. Loop tiling for GPUs can improve coalesced global-memory accesses, data reuse in local memory, and reduce the effect of thread divergence [GCH+14].

Manually tuning a kernel to apply these optimisations is a complicated process and it requires additional modifications when the input program changes. Lines 8 and 9 of Listing 5.1 highlight the need for explicitly allocating arrays into the local memory region of the GPU with a statically defined size. This size is a key factor for performance as it needs to match the work-items defined in each work-group (i.e., local workgroup size). Moreover, lines 15-19 illustrate how data is being copied from the global memory into smaller chucks into the local memory. This process requires complex indexing usage through explicit attributes (e.g., `get_group_id()`). Lines

```
1  __kernel void GEMM(int M, int N, int K, _global float* A,
2                     __global float* B,_global float* C) {
3   int row = get_local_id(0);
4   int col = get_local_id(1);
5   int globalRow = TileSize*get_group_id(0) + row;
6   int globalCol = TileSize*get_group_id(1) + col;
7
8   __local float Asub[TileSize][TileSize];
9   __local float Bsub[TileSize][TileSize];
10
11  float acc = 0.0f;
12
13  const int numTiles = K/TileSize;
14  for (int t=0; t<numTiles; t++) {
15   int tiledRow = TileSize*t + row;
16   int tiledCol = TileSize*t + col;
17
18   Asub[col][row] = A[tiledCol*M + globalRow];
19   Bsub[col][row] = B[globalCol*K + tiledRow];
20
21   barrier(CLK_LOCAL_MEM_FENCE);
22
23   for (int k=0; k<TileSize; k++) {
24    acc += Asub[k][row] * Bsub[col][k];
25   }
26
27   barrier(CLK_LOCAL_MEM_FENCE);
28  }
29
30  C[globalCol*M + globalRow] = acc;
31 }
```

Listing 5.1: Example of an OpenCL kernel computing a Matrix Multiplication optimised to use local memory and loop tiling.

23-25 showcase how the computation is performed in the newly introduced tiled loop. This computation iterates over a loop of a specific size (i.e., tile size) which needs to be tuned for the data to fit in the local memory. Also, this tiled loop needs to be explicitly synchronised with the communication barriers highlighted in lines 21 and 27. Therefore, the process outlined above highlights the need for in-depth understanding of the underlying GPU architecture, as well as the OpenCL programming model to exploit the memory hierarchy of GPUs.

Listing 5.2 showcases a Matrix Multiplication method written in Java with the

```
private static void matrixMultiplication(final float[] A,
 final float[] B, final float[] C,
 final int size) {
  for (@Parallel int i = 0; i < size; i++) {
      for (@Parallel int j = 0; j < size; j++) {
          float sum = 0.0f;
          for (int k = 0; k < size; k++) {
              sum += A[(i * size) + k] * B[(k * size) + j];
          }
          C[(i * size) + j] = sum;
      }
  }
}
```

Listing 5.2: Example code of a Java method computing a naive implementation of a Matrix Multiplication written with the TornadoVM API.

TornadoVM API. This example highlights that the developer is only required to use the @Parallel annotation to mark the parallel dimensions of a loop in a given method. In addition, the underlying complexity is being handled by the TornadoVM JIT compiler.

The original TornadoVM framework did not utilise local memory. Prior to this work, for Java applications to merit from local memory in GPUs typically relied on exposing low-level programming primitives [SCN+15] to the API. The goal of the work presented in this Chapter is to preserve intact the minimal API of TornadoVM, and exploit the optimisations mentioned above seamlessly during JIT compilation.

5.2 GPU Memory-Aware JIT Compilation

Figure 5.1 presents an overview of the JIT compilation process for exploiting local memory. The underlying approach includes three distinct phases: *detection*, *compositional intrinsics*, and *memory transformations*. All transformations are applied to the common IR of TornadoVM JIT compiler (which is a superset of the Graal IR [DSW+13]). The TornadoVM IR uses the sea-of-nodes [CP95] common representation which encompasses both the control-flow and data-flow nodes. By using this representation, TornadoVM can compile and optimise Java bytecodes to OpenCL by performing IR-node substitutions, including the addition or deletion of nodes through the compilation process.

The ***detection*** phase scans the IR to locate specific nodes, such as accesses to/from arrays through read and write nodes, as well as the induction variables. To use local

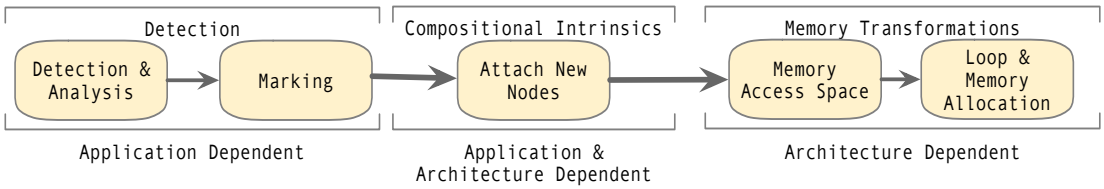


Figure 5.1: Overview of the Just-in-Time compilation flow for automatically exploiting the GPU memory hierarchy.

memory, nodes commonly used to read and write from/to memory are first located (by default, the JIT compiler assumes all accesses target the global memory). These array accesses are detected via indexed read and write nodes in the IR. The detection phase is crucial for the compilation process since:

(a) It provides the exact region in the IR to read/write from/to local, instead of global memory.

(b) It analyses all nodes accessible from the indexed read/write nodes, such as the induction variables and the parameters of the compiled method.

This information is accounted during the detection phase to introduce and attach a new node. The newly introduced node encloses the read and write nodes, and it is used by the next phases to perform aggressive optimisations regarding local memory (Sections 5.4) and loop tiling (Section 5.4.2).

The ***compositional intrinsics*** phase adds to the IR the nodes needed for performing memory allocation, and prepares the IR for code generation. Concisely, this phase starts by specialising the IR with GPU-oriented optimisations based on the new nodes trailing from the detection phase. Although the previous phase was only application dependent, from this stage and onwards application and architecture-dependent optimisations are being applied to the IR. During this process, the high-level IR is lowered into a more concrete lower IR (known as lowering process) which has a closer mapping to the underlying target architecture. Since this process involves the introduction of several new IR nodes, a key design decision was to create and utilise a set of parameterised compiler intrinsics. In this way, complex optimisations can be decomposed (e.g., loop tiling with local memory) into smaller graphs, and combined at runtime to form larger graphs. These compiler intrinsics are in the form of *Snippets* [SWU+15] and they are methods, completely written in Java, which represent low-level operations that are being attached to the IR at runtime. Examples of such operations are the creation of local memory allocation code or the combination of loop tiling with the local memory

allocation. Section 5.3 explains in detail, all the compiler intrinsics introduced for automatically supporting local memory.

Finally, the ***memory transformations*** phase is an architecture dependent optimisation process. In this phase, the JIT compiler processes the new nodes introduced during lowering, and completes the IR by adding the correct information to access local memory. This low-level information includes the base addresses and the offset arithmetic nodes. In summary, this phase introduces new IR operations for the following:

1. Copying data from global to local memory.
2. Materialising the indices to read/write from/to local memory.
3. Copying the final data (i.e., the variable that escapes the scope of the loop) from local to global memory upon finishing executing a kernel.

In addition, this phase invokes the OpenCL API for obtaining device-specific information to optimise local memory sizes based on the number of work-items deployed and the available local memory.

5.3 Compositional Compiler Intrinsics (CCIs)

Compiler intrinsics are low-level code segments typically expressed in low-level programming languages, such as Assembly or C. They represent optimised code for common operations, such as the use of vector operations or memory allocation. The JIT compilers of Graal and MaxineVM [WHVDV+13, KCR+17b] introduced the concept of *compiler snippets* as a high-level representation of low-level operations [SWU+15]. With snippets, low-level operations are implemented in a high-level programming language (Java) instead of the assembly code. Since the aforementioned JIT compilers are also implemented in Java, they do not need to cross language boundaries to implement their intrinsics. Hence, their code can be further optimised by applying common compiler optimisations (e.g., loop unrolling, constant propagation, etc.).

Fumero *et al.* [FK18] extended the use of compiler snippets to express efficient parallel skeletons for GPUs in TornadoVM. The work in this chapter extends the capabilities of compiler snippets to express local memory optimisations by introducing *compositional compiler intrinsics*, which can be parameterised and reused for different compiler optimisations. With this approach, applications can further increase their performance by automatically exploiting local memory.

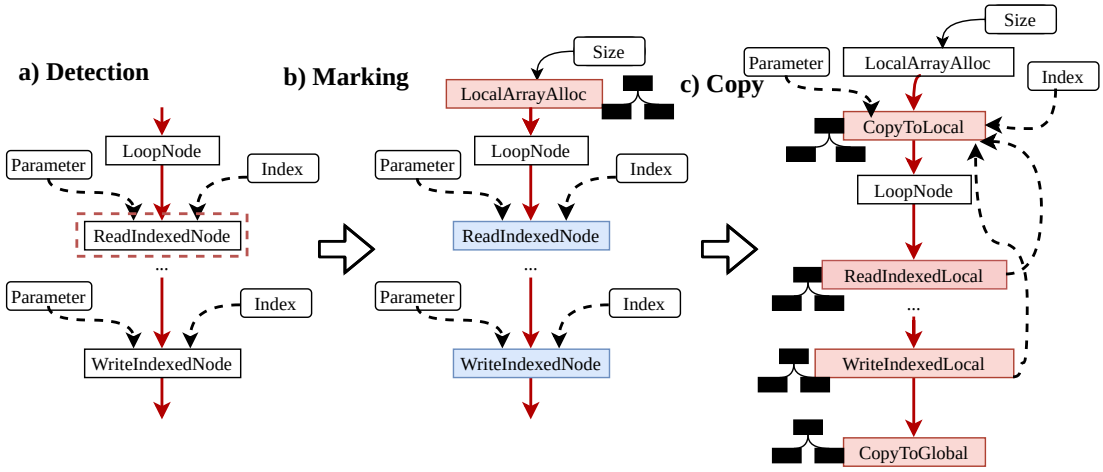


Figure 5.2: IR transformations for the compiler intrinsics of local memory allocation and data copies.

Key focus of this work is to provide a set of parameterised compiler intrinsics that gradually lower the IR, and generate efficient GPU code that uses local memory. These compiler intrinsics are involved in two different compilation phases: the *compositional intrinsics* phase (Figure 5.1), in which the actual compiler intrinsics are inserted into the compiled graph (IR), and the *memory transformations* phase, in which the IR is optimised after inlining the intrinsics into the graph. This approach offers a degree of flexibility to the compiler to apply several optimisations and combine intrinsics to express multiple optimisations. In detail, the following intrinsics are introduced:

- *Local Memory Allocation:* This intrinsic modifies the IR to emit code for allocating arrays in local memory. Input and output variables that have been detected in the detection phase are marked as candidates for using local memory. In this case, this compiler intrinsic introduces the logic to declare and instantiate arrays in local memory. By design, snippets do not support dynamic memory allocation, and consequently, the *Local Memory Allocation* intrinsic does not either. Therefore, array lengths must be statically set. To address this limitation, the lengths of the arrays are provided to be stored in local memory as a parameter node that can be dynamically changed and updated in the *memory transformations* phase. The actual size depends on the amount of local memory available on the target device and the number of threads deployed. In this way, multiple combinations of local memory sizes can be generated during runtime. Figure 5.2 illustrates the use of this compiler intrinsic for loop tiling in the JIT compiler. The left-hand side of Figure 5.2 shows the IR that represents an indexed read and

an indexed write from/to an array inside a loop. The graph is read as follows: the control flow nodes are connected with red arrows, while the data-flow nodes are connected with black dashed arrows. Moreover, the introduction of a compiler intrinsic is represented by a red node, while a blue node represents a required node to perform an optimisation. In this phase, the JIT compiler runs the detection phase, looking for reads and writes enclosed in loops. Upon the detection of the `ReadIndexedNode`/`WriteIndexedNode` nodes (Figure 5.2(a)), the compiler marks them as candidates to use local memory and introduces a set of new nodes (i.e., `LocalArrayAlloc`, `Size`) in the IR (Figure 5.2(b)).

- *Copy To Local Memory/Copy To Global Memory:* These compiler intrinsics introduce a copy from global to local memory and vice versa. These memory copies are presented in Figure 5.2(c) as two new IR intrinsics `CopyToLocal` and `CopyToGlobal`. Both intrinsics are performed during the *memory transformations* phase and accept as inputs the local array nodes and the corresponding indices from global and local memory.

- *Load/Store Operations in Local Memory:* This pair of intrinsics performs load and store operations from arrays that reside in local memory to private memory, and vice versa. Figure 5.2(c) illustrates these operations as two new IR intrinsics `ReadIndexedLocal` and `WriteIndexedLocal` that represent the load and store operations, respectively. This pair of compiler intrinsics enables the JIT compiler to access the local memory address space, as opposed to the TornadoVM IR indices (`ReadIndexed` and `WriteIndexed` in Figure 5.2(b)) that do not support this functionality.

- *Reductions with local memory:* This intrinsic improves the reduction operations presented by Fumero *et al*. [FK18], by adding local memory support. Using the same technique as described for the two previous compiler intrinsics, the GPU local memory is utilised to increase the performance of reduction operations on GPUs. Section 5.4 explains all the IR transformations involved to generate efficient GPU reductions using local memory via the compiler intrinsics.

- *Parameterised Loop Tiling for Local Memory:* A set of compiler intrinsics that can be combined with common loop optimisations, such as loop tiling and loop unrolling, are introduced. Although these loop optimisations are orthogonal to local memory, they can facilitate the use of local memory. To do so, a compiler

intrinsic in the JIT compiler to perform loop tiling is added. This intrinsic receives, as parameters, all arrays references stored in local memory and all loop indices that access local memory. Through the parameterised architectural design of the compiler intrinsics, this optimisation can be combined with loop unrolling. Figure 5.5 illustrates an example of this compiler intrinsic that combines local memory allocation with loop tiling. Figure 5.5(a) shows the detection phase with three primary nodes: a loop node and two indexed read and write nodes. During the detection phase, the loop node is selected as a candidate node for loop-tiling. The second graph shows the expansion of the IR through the introduction of the compiler intrinsic for loop-tiling. This new set of nodes in the IR enables a new marking phase to apply local memory and loop tiling (Figure 5.5(b)). Figure 5.5(c) shows the new IR after applying local memory allocation, loop tiling, and the copies from global to local memory (and vice versa once the loop tiling optimisation is performed).

5.4 Exploiting Local Memory

To demonstrate the outlined technique, two different use cases were used. These use cases showcase how compositional compiler intrinsics are introduced in the IR and how they are optimised to efficiently utilise the GPU memory hierarchy. Although this approach is developed in the context of the JIT compiler in TornadoVM, the technique can be used by other compilation frameworks that provide similar features, such as LLVM [LA04] and GCC [SD09].

5.4.1 Parallel Reductions

The first use-case that aims to showcase the optimisation process is a set of reduction operations, which are defined as the accumulation of input values from a vector into a single scalar value. Reduction operations are widely used algorithmic parallel skeletons [Col91]. In addition, there are a key building block for many parallel programming frameworks, such as Google Map/Reduce [DG08], Apache Spark [ZXW+16], Apache Flink [KS16], and common libraries, such as Thrust [BH12]. Therefore, optimising parallel reductions has been a well-studied topic, especially regarding memory optimisations, such as local memory [DGHGL+19, CA12].

Figure 5.3 illustrates a basic technique for a reduction operation from the OpenCL

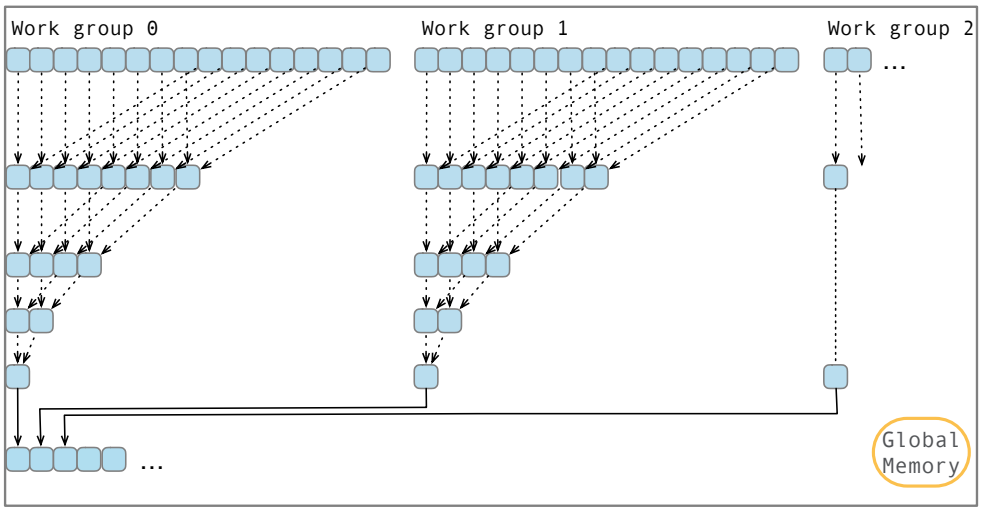


Figure 5.3: GPU-targeted reduce operation with *explicit* global memory allocation.

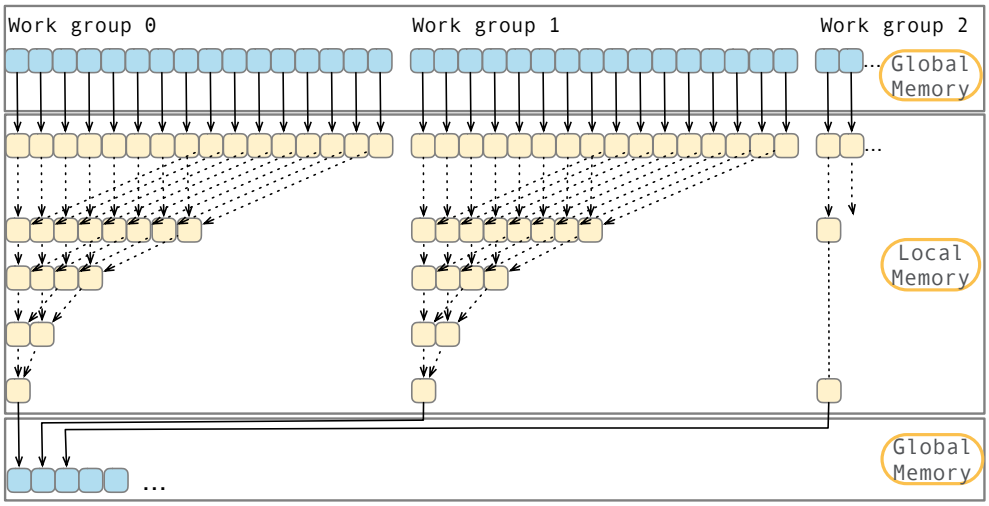


Figure 5.4: GPU-targeted reduce operation with local memory allocation.

perspective. On the contrary, Figure 5.4 illustrates a basic technique to use local memory for partial copies to optimise performance for reduction operations. A key aspect of this technique relies on making copies of data batches per work-group to perform individual reductions and return a single reduced value. Hence, performance is significantly improved, since accesses to the main global memory are reduced.

To perform high-performance reductions on GPUs, TornadoVM currently uses compiler intrinsics [SWU+15] to express parallel skeletons [FK18]. TornadoVM already solves the problem of seamlessly expressing parallel reductions in the compiler, albeit without exploiting data locality and GPU local memory. Each compiler phase for adding

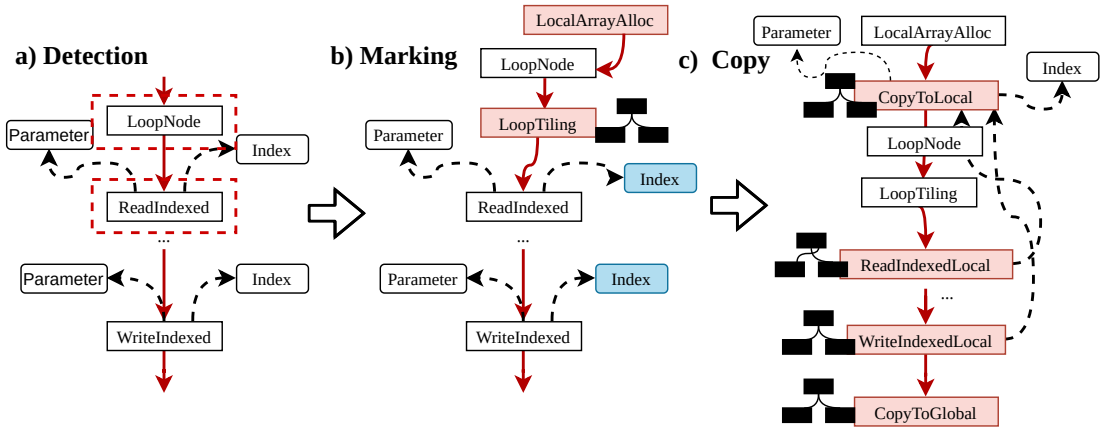


Figure 5.5: IR transformations for the compiler intrinsics of loop tiling, local memory allocation, and data copies.

local memory support to reduction operations is described below.

Detection To express reductions in TornadoVM, developers can use the *@Reduce* annotation as demonstrated in Listing 5.3. Upon adding the annotation, the TornadoVM JIT compiler detects the reduction pattern, which is subsequently used to add local memory support.

Briefly, the version of TornadoVM (v0.5) used only targets the global memory space by automatically dividing the iteration space into smaller chunks (one chunk per work-group), and it performs a full reduction within each chunk. This strategy is illustrated in Figure 5.3.

Lowering TornadoVM implements parallel reductions with intrinsics (further information can be found in [FK18]). Listing 5.4 exemplifies the compiler intrinsic (snippet) that TornadoVM uses to perform the reduction operation of Listing 5.3. As shown, the compiler intrinsic is also written in Java, and during compilation, its generated IR is appended to the rest of the IR graph of the original input method. Consequently,

```
1 public void reduce(float[] in, @Reduce float[] out) {
2  for (@Parallel int i = 0; i < n; i++) {
3   out[0] += in[i];
4  }
5 }
```

Listing 5.3: Example code of a Java reduction written with the TornadoVM API.

```
1  @CompilerIntrinsinc
2  void reductionIntrinsic(float[] input, float[] output){
3    int idx = OpenCL.get_local_id(0);
4    int lgs = OpenCL.get_local_size(0);
5    int gID = OpenCL.get_group_id(0);
6    float[] local = OCL.alloc(SIZE, float.class);
7    local[idx] = input[OpenCL.get_global_id(0)];
8    for(int i = (lgs/2); i > 0; i/=2) {
9      OpenCL.localBarrier();
10     if (idx < i) local[idx] += local[idx + i];
11   }
12   if (idx == 0) output[gID] = local[0];
13 }
```

Listing 5.4: Example code of a compiler intrinsic to utilise the GPUs local memory for reductions.

the merged IR can be re-optimised iteratively, a key advantage compared to intrinsics written in low-level languages that are treated as native functions by the compiler.

The existing intrinsic has been augmented in order to add local memory support as shown in Listing 5.4 (gray colour). To achieve this, additional compiler intrinsics have been added, to express local memory regions in a high-level manner. In this case, the use of local memory region is explicit by allocating the corresponding arrays in the generated OpenCL source code instead of defining a parameter to the generated OpenCL kernel with a local memory region. Line 6 shows the allocation of the `local` array in local memory. This allocation is performed via an invocation to the static method `OCL.alloc`, in which the size and the type of the array are passed along. Consequently, line 7 copies data from global memory to local memory. Then, the actual reduction is computed using local memory (line 10). Finally, line 12 performs the final copy from local to global memory. The JIT compiler lowers these intrinsics to generate OpenCL C code that corresponds to the high-level Java code. By using this strategy of computing with local memory, the execution flow of Figure 5.3 is transformed to that of Figure 5.4.

During the lowering phase, the generated compiler graph includes new nodes associated with allocating, indexing, and storing data to the local memory region. Then, the new nodes are inlined to the IR graph of the compiled method. Figure 5.6 depicts the IR transformations upon replacing the IR nodes introduced by the intrinsic in Listing 5.4 with the corresponding lowered IR nodes (via substitution) for local memory allocation. Similarly to Figure 5.2, control-flow nodes are connected with red arrows, while data-flow nodes are connected using black dashed arrows. The left graph in

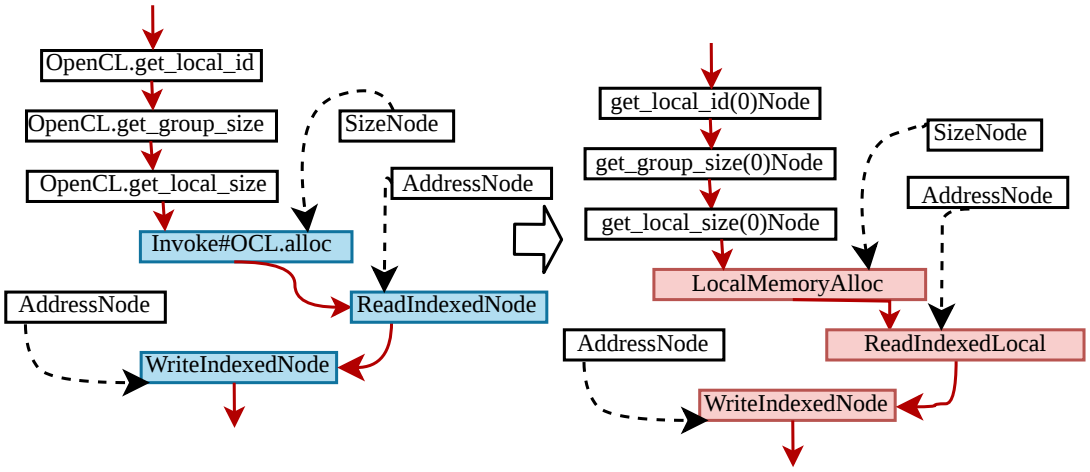


Figure 5.6: Node replacements during the lowering phase for the reduction compiler intrinsic.

Figure 5.6 represents the IR when the code for the reduction intrinsic is built. This graph includes the `Invoke#OCL.alloc` node representing an array allocation using local memory. This node contains information about the size used as a data-flow node, allowing us to dynamically change the size. Therefore, the same compiler intrinsic can generate parameterisable code for various local memory sizes. The right graph shows the IR graph after applying the substitution to allocate local memory. As shown, the output array of `Invoke#OCL.alloc` has been replaced by a `LocalMemoryAlloc` node. Since this type of nodes only appears in intrinsics related to reductions, TornadoVM guarantees that these intrinsics only substitute local arrays.

Memory Transformations A challenge in this phase is that the decision for the statically allocated size of local memory. This phase must consider the deployed GPU threads (work-items) along with available local memory size to avoid memory aliasing. However, the number of deployed threads is determined at runtime and depends on the input data size for each application. To tackle this challenge, the sizes of the local arrays are attached as a data-flow node (`SizeNode`) in the IR, as illustrated in Figure 5.6. In this case, if the same reduction is executed, during run-time, with different input size the generated code will be dynamically adapted by changing only the size node that is attached to the `LocalMemoryAlloc` node in the IR.

```
@CompilerIntrinsinc
void tile(float sum, float[] arrA, float[] arrB, int size,
 ValueNode operator, ValueNode reduceOperator) {
  OpenCL.localBarrier();
  for (int x = 0; x < size; x++) {
    sum = OCL.compute(arrA[x], arrB[x],
              operator, reductionOperator);
  OpenCL.localBarrier();
  }
}
```

Listing 5.5: Example code of a compositional compiler intrinsic for processing loop tiling using local memory.

5.4.2 Matrix Operations

The second use-case that illustrates the efficacy of the extensions to the JIT compiler is an $O(N^3)$ matrix multiplication operation. Previously, in Listing 5.2 of Section 5.1 an example of a matrix multiplication written with the TornadoVM API was presented. This code has three nested loops that can be parallelised via TornadoVM by employing the *@Parallel* annotation. This section explains all the phases in the JIT compilation flow that facilitate data locality in the local memory.

Detection The detection phase of the JIT compiler traverses the IR graph seeking the `ReadIndexed` and `WriteIndexed` nodes, which represent the memory accesses to the global memory. Figure 5.5(a) illustrates this process in which all the derived information about the induction variables and the parameters of the method is combined. This process contributes to the addition of two new nodes that apply two compiler intrinsics; one for local memory allocation and a second for loop tiling at the innermost loop.

Lowering The compiler marks the first loop entry for each input array to introduce the compiler intrinsics for the local memory allocations. Additionally, the compiler applies the loop tiling optimisation to detect the three nested loops corresponding to the matrix multiplication application. Figure 5.5(b) presents the marking of the two nodes that were added in the previous phase (`LocalArrayAlloc` and `LoopTiling`). During the lowering phase, the IR nodes are replaced by the respective compiler intrinsics. Previously, Section 5.3 discussed the local memory allocation intrinsic. Now, Listing 5.5 shows the code that implements the compiler intrinsic in the JIT compiler for loop

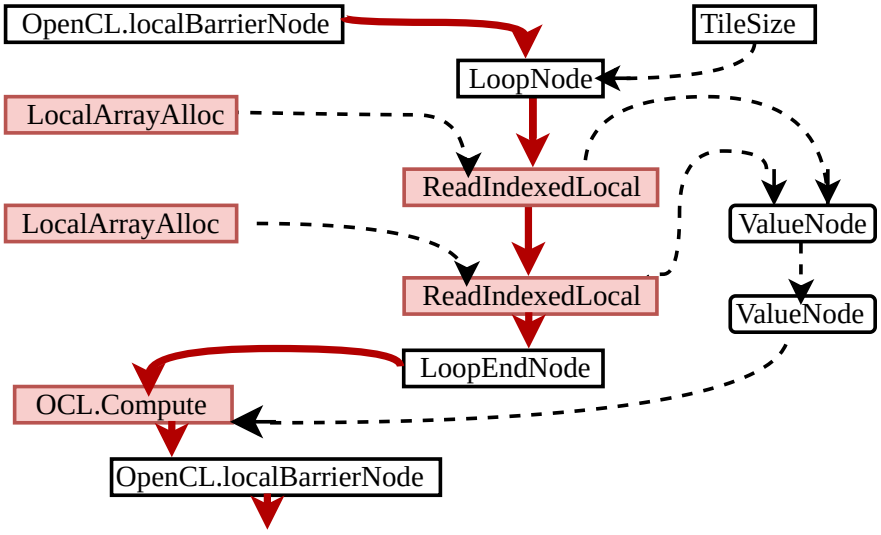


Figure 5.7: IR nodes from the compiler intrinsic in Listing 5.5.

tiling. This intrinsic accepts as inputs a set of arrays, the size for the loop tiling, and the operators to be applied inside the loop tiling. Line 5 shows the new loop that performs the tiling, and line 6 shows a method invocation that introduces the compute logic inside this new loop. Also, two OpenCL local barriers are required to guarantee consistency. The first barrier in line 5 is used before loop tiling to ensure that the data has been copied to the allocated space in local memory. Furthermore, the second barrier (line 8) synchronises the tile processing across all work-items before the final copy to the global memory. Moreover, developers do not need to account for about maintaining memory consistency when using local memory since the JIT compiler automatically inserts the barriers. Figure 5.7 shows the IR representation for this compiler intrinsic. The new loop is introduced as a control flow node (`LoopNode`) right after the OpenCL local barrier node. These barrier nodes are lowered from the compiler intrinsic, and thus it is ensured that the number will be correct. The loop body is represented by a compiler intrinsic called `OCL.Compute`. This intrinsic acts as a placeholder for inserting the IR nodes representing the core computation within the loop tiling, which, in the case of matrix multiplication, corresponds to a multiplication followed by a sum. In turn, all these new nodes will be replaced during the memory transformations phase.

Memory Transformations Figure 5.8 illustrates the transition of the IR from lowering (left graph) to the final memory transformations phase (right graph). In the last phase before the OpenCL C code generation, a set of new compiler intrinsics

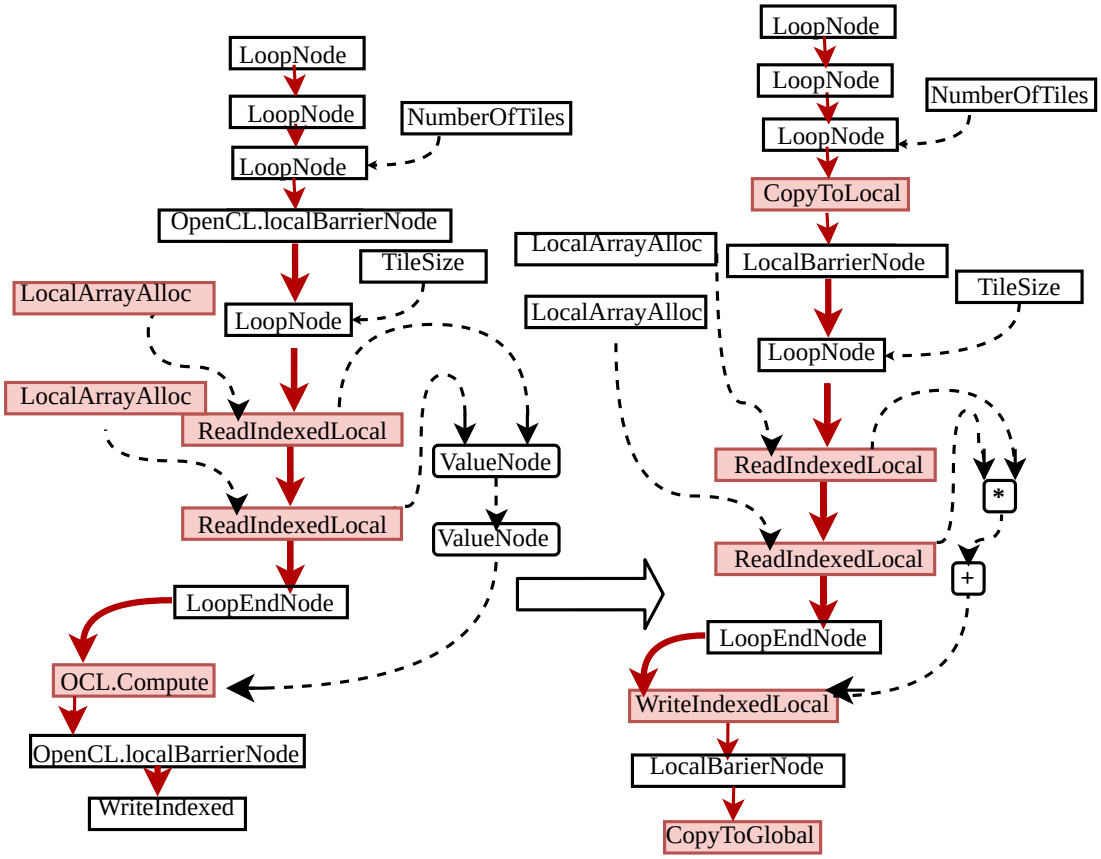


Figure 5.8: IR node replacements during the memory transformation phase for the Matrix Multiplication application.

(e.g., `CopyToLocal`, `CopyToGlobal`) is introduced to use local memory. The `WriteIndexedLocal` intrinsic of the right graph in Figure 5.8 is used to store the result from the `sum` variable (Listing 5.5 - line 7). This phase has been previously discussed in Section 5.4.1.

Regarding the loop tiling optimisation, the left graph in Figure 5.8 shows the IR of the loop tiling compiler intrinsic (Figure 5.7). During the *memory transformations* phase, all the IR nodes of the compiler intrinsics are lowered to OpenCL instructions. This phase inlines the call of the `OCL.Compute` method that it was introduced in the previous phase into a set of nodes. In this case, the call inlines all nodes involved in the matrix multiplication operation within the loop tiling (see right graph in Figure 5.8).

Since the loop tiling compiler intrinsic is applied to the innermost loop, three more nodes (`LoopNode`) are illustrated in Figure 5.8 representing the three outermost loops. Therefore, the lowering process of loop tiling starts by first traversing the IR graph from the innermost loop and replacing its loop bound with a `TileSize` node and the bounds

of the third innermost loop with a `NumberOfTiles` node. The two outermost loops remain the same as they represent the sizes of parallel dimensions. To decide the tile size during JIT compilation, the OpenCL driver is invoked to provide the maximum number of the available work-items, which is device-specific. Similarly, the number of deployed threads (`GlobalWorkItems`) is obtained from the OpenCL driver as it matches the input data size of the given application. This information is used to calculate the number of total tiles.

Finally, due to the parameterisable compiler intrinsics, existing compiler intrinsics can be combined with more aggressive optimisations, such as loop unrolling and partial escape analysis.

Although compiler intrinsics, in the form of snippets, offer an expressive and low-overhead mechanism to define memory optimisations, they generate less optimal code compared to aggressive optimisation approaches, such as polyhedral compilation. The key difference is that the proposed approach does not explore the optimisation space for the best possible combination for tile sizes, amount of local memory, and work-item sizes. However, if the methodology is combined with TornadoVM dynamic reconfiguration mechanism [FPZ+19], a similar approach to polyhedral compilation could be achieved dynamically at runtime. Also, other approaches, such as those proposed by Cummins *et al*. [CPSL15, CPWL17] and by Wang *et al*. [WO18] indicate that Machine Learning can assist to improve the quality of the generated code.

5.5 Evaluation

This section presents the performance evaluation of the proposed optimisations along with the compilation overhead induced to the existing JIT compiler.

5.5.1 Experimental Setup and Methodology

5.5.1.1 Experimental Methodology

Table 5.1 presents the hardware specifications of the three GPU devices used in the experimental setup. The system runs CentOS 7.4 with Linux kernel 3.10, and for all experiments, the OpenJDK JVM 1.8 (u242) 64-Bit with 16GB of Java heap memory was used. To ensure that the JVM has been warmed up, 100 warm-up iterations per benchmark were performed, and only the geometric mean of the next 100 runs is reported. Table 5.2 outlines the exact specification of GPUs used in the evaluation.

Table 5.1: Experimental setup and configuration.

Hardware	
Processor	Intel Core i7-9750H CPU @ 2.60GHz
Cores	6 (12 HyperThreads)
RAM	32GB
Integrated GPU	Intel UHD Graphics 630
Discrete GPU [1]	NVIDIA GeForce GTX 1650 (Turing) 4GB GDDR5, 896 CUDA Cores
Discrete GPU [2]	AMD GFX900 4GB GDDR5, 896 CUDA Cores
Software	
Operating System	CentOS 7.4 (Kernel 3.10-generic)
OpenCL (CPU)	2.1 Device Version
OpenCL (IGPU)	2.1 Device Version
OpenCL (GPU)	1.2 Device Version
CUDA Driver	450.80.02
TornadoVM	v0.7
JVM	OpenJDK JVM 1.8 (u242) 64-Bit
Java Heap	-Xmx16G -Xms16G

Table 5.2: GPU configuration: Device, memory, work-item and driver specification.

GPU	**Vendor**	**Work-Items**	**Global**	**Local**	**Driver**
GFX900	AMD	1024x1024x1024	8GB	64KiB	2766.4
GeForce 1650	Nvidia	1024x1024x64	4GB	48KiB	435.21
HD Graphics	Intel	256x256x256	25GB	64KiB	19.43.14

5.5.1.2 Benchmarks and Input Sizes

The baseline implementations are the following: (i) the original code produced by TornadoVM[1] that does not exploit GPU local memory, and (ii) the hand-written optimised OpenCL code, of the functionally equivalent OpenCL benchmarks. The OpenCL baseline implementation includes the same set of optimisations as the extended JIT compiler.

In addition, the proposed technique is evaluated against three reduction operations

[1]The exact commit point is: `https://github.com/beehive-lab/TornadoVM/commits/81c70437800c252899a56e78ddbe80697f273973`.

Table 5.3: List of benchmarks used for the evaluation of the extensions to the JIT compiler.

Benchmark	Input Sizes	Method/Kernel LOC			Optimisations	
		Java	Gen	OpenCL	local Memory	Loop Tiling
Reduction (Min)	2^8 to 2^{24}	5	40	19	✓	✗
Reduction (Add)	2^8 to 2^{24}	5	40	19	✓	✗
Reduction (Mul)	2^8 to 2^{24}	5	40	19	✓	✗
Transpose Matrix	2^8 to 2^{24}	6	77	14	✓	✗
Matrix Multiplication	2^5x2^5 to 2^{12}x2^{12}	11	63	25	✓	✓
Matrix Vector Multiplication	2^6x2^3 to 2^{16}x2^8	9	55	20	✓	✓

(*Minimum*, *Addition*, and *Multiplication*), and three matrix operations (*Matrix Multiplication*, *Matrix Transpose*, and *Matrix Vector Multiplication*). Table 5.3 presents the various parameters used for each benchmark, including the input data size, the lines of code (LOC), and the combination of optimisations applied per benchmark; namely Local Memory usage and Loop Tiling. The evaluated benchmarks have been implemented in Java for execution with TornadoVM and in OpenCL C for comparisons against hand-written optimised native code. The third column (Java) of Table 5.3 shows the LOC of the TornadoVM-Java implementations, while the fourth column (Gen) shows the LOC of the auto-generated GPU code. Finally, the fifth column (OpenCL) shows the LOC of the manually written OpenCL C codes. The LOCs of the implementations is order to provide an insight into the complexity of the developed code with respect to utilising the GPU memory hierarchy. In addition, the OpenCL code generation in TornadoVM (Gen) derives from SSA (Static Single Assignment) representation (in which each operation is assigned exactly once). Therefore, more lines of code are generated. Regarding optimisations, all reductions exploit local memory as explained in Section 5.4, whereas the matrix operations exploit the different combinations (local Memory, Loop Tiling), as discussed in Section 5.4.2.

5.5.2 Performance Evaluation

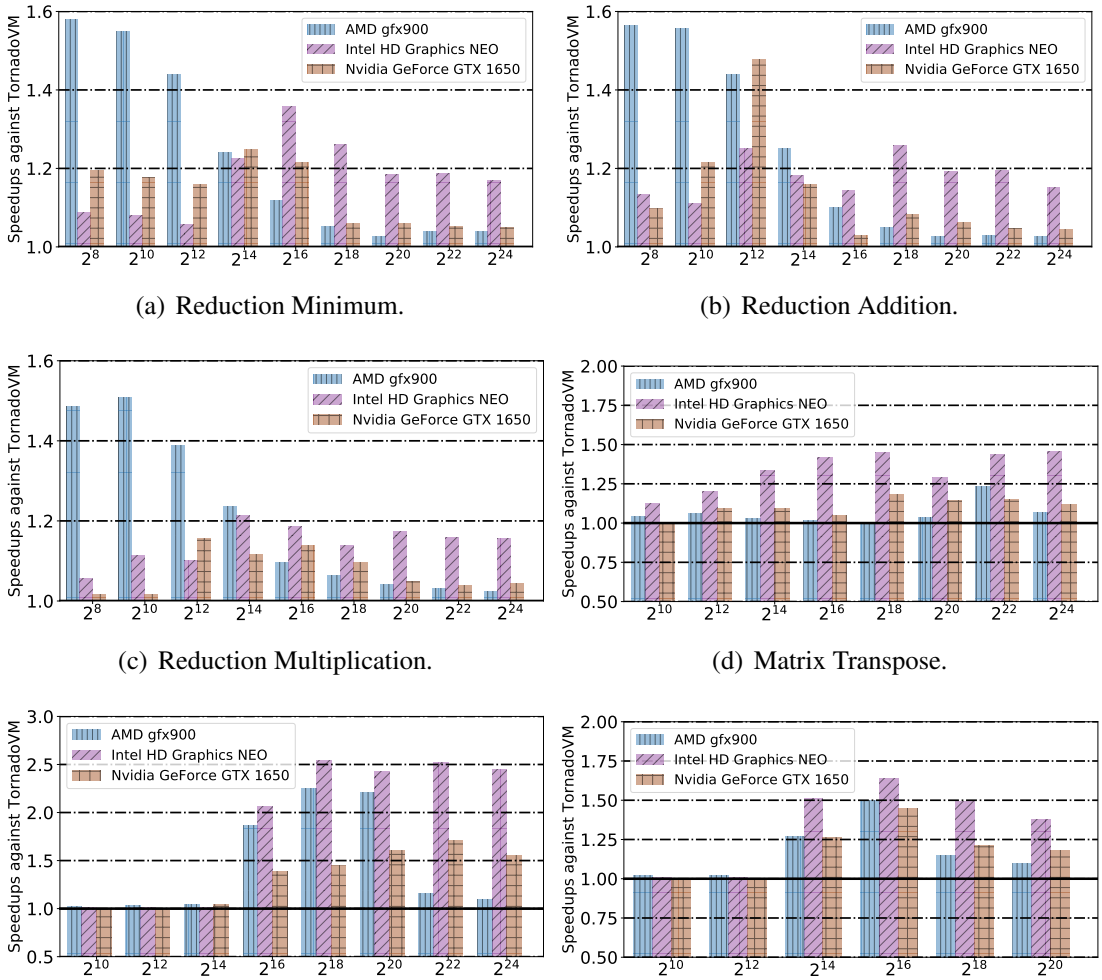
5.5.2.1 Performance Comparison against TornadoVM

Figure 5.9 presents the performance improvements achieved by the compiler optimisations, against TornadoVM which does not support local memory. For all of the figures, the x-axis shows the input size for each benchmark, while the y-axis shows the achieved speedup against TornadoVM.

In general, the proposed approach outperforms TornadoVM by up to 2.5x and 1.6x for matrix and reduction operations, respectively. Additionally, all benchmarks exhibit performance speedups across all data sizes. For Intel and Nvidia GPUs, the reported times include only the kernel execution on the GPUs. On the contrary, for the AMD GPU, the reported times also include data transfers. This is due to a limitation of the AMD OpenCL driver, which can only report kernel execution and data transfer times combined. For this reason, the discussion regarding performance between the different GPUs is divided into two separate paragraphs.

Performance on an AMD GPU: As illustrated in Figure 5.9, the compiler optimisations yield performance speedups ranging from 1.02× to 1.58× on the AMD GPU. Regarding all reduction operations (Figure 5.9(a-c)), one can observe that the execution for small input data sizes yields higher performance compared to larger input sizes when utilising local memory (up to 1.58x at 2^8 data elements in Figure 6.9(a)). Since the AMD GPU reported times also include data transfers, the observed speedups degrade as the input data sizes increase due to the costly data transfers. Nevertheless, these overheads do not result in slowdowns. Regarding matrix operations (Figure 5.9(d-f)), the execution on the AMD GPU obtains a maximum performance of 2.3x for matrix multiplication and 1.23x for matrix transpose, following similar trends with the reduction operations.

Performance on Nvidia and Intel GPUs: As shown in Figure 5.9, the execution with local memory on Intel HD Graphics (second bars) performs up to 35% faster than the baseline configuration (2^{16} data elements in Figure 6.9(a)). Regarding the execution on the Nvidia GPU (third bars), performance improvements of up to 48% are observed (2^{12} data elements in Figure 5.9(b)). As the data sizes increase, the relative performance speedups due to the optimisations decrease. This is attributed to the additional global barrier that had to be placed into the generated code before the final read from local to global memory. As the number of threads increases and surpasses the number of



(a) Reduction Minimum.

(b) Reduction Addition.

(c) Reduction Multiplication.

(d) Matrix Transpose.

(e) Matrix Multiplication.

(f) Matrix Vector Multiplication.

Figure 5.9: Performance comparison against vanilla TornadoVM (*x-axis:* Input sizes in powers of 2, *y-axis:* Achieved speedup).

physical threads that can run in parallel on the device, the overhead due to the barriers also increases. To address the barrier overhead, node hoisting will be applied in future work.

Concerning matrix operations (Figure 5.9(d-f)), the largest speedup (up to 2.5x) is observed when running on the Intel HD Graphics (2^{18} data elements in Figure 5.9(e)). In general, the observed speedups for matrix operations are higher than those in reduction operations, due to the combination of the applied optimisations (i.e., loop tiling and local memory). Finally, as shown in Figure 5.9(e-f), for small data sizes, no performance improvements were observed. This is attributed to the loop unrolling optimisation taking place at the early stage of the optimisations, which negates the impact of allocating to

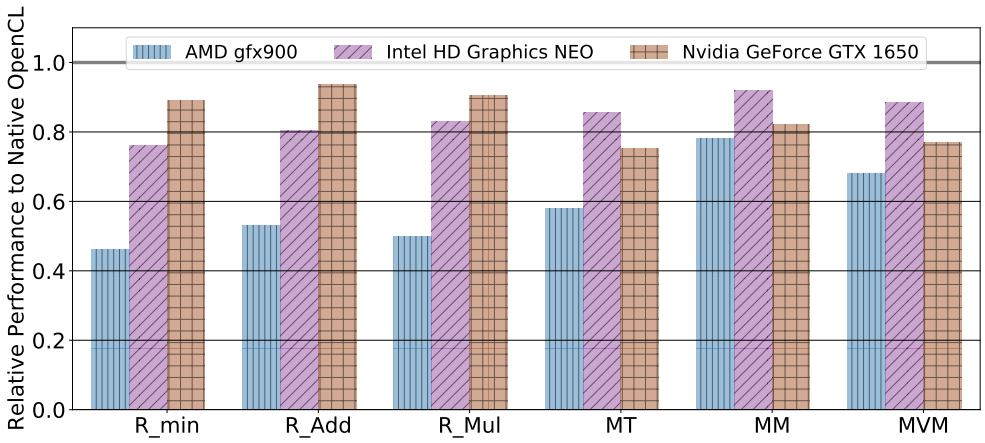


Figure 5.10: Relative performance of the code generated through the extended JIT compiler against hand-written optimised OpenCL implementation (the higher, the better).

data to the local memory. Nevertheless, it is possible to apply further optimisations on unrolled loops in future work.

5.5.2.2 Performance Comparison against Hand-Written OpenCL

Figure 5.10 shows the relative performance of the code generated by the JIT compiler against the functionally equivalent optimised (using local memory and loop tiling) OpenCL code. Similarly, to the previous experiments, the times reported on the AMD GPU include both kernel and data transfer times in contrary to Intel and Nvidia GPUs that report only kernel times.

As shown in Figure 5.10, the performance of the JIT-compiled code compared to native OpenCL C implementations for reductions, reaches up to 53% on the AMD GPU, up to 83% on the Intel HD GPU, and up to 94% on the Nvidia GPU. Regarding matrix operations (Figure 5.10), the JIT-compiled code performs up to 78% on the AMD GPU, up to 92% on the Intel HD GPU, and up to 82% on the Nvidia GPU, compared to the native OpenCL C code. These results indicate that the auto-generated code performs competitively, if one take into account that no user intervention for performance tuning is required.

5.5.3 Compilation Overhead

Table 5.4 presents the time spent for JIT compilation separated into two categories: TornadoVM and driver compilation times. The compilation time in TornadoVM is the

Table 5.4: Compilation times per compilation stage in the two stage compilation process.

Time in Milliseconds (*ms*)				
Benchmark	**TornadoVM**	**Nvidia Driver**	**Intel Driver**	**AMD Driver**
Reduction Addition	64.59	47.04	224.38	18.54
Reduction Multiplication	73.23	54.60	251.16	19.64
Reduction Minimum	81.38	57.70	258.61	18.85
Matrix Transpose	55.43	43.20	227.73	17.42
Matrix Multiplication.	62.21	48.10	250.68	21.32
Matrix-Vector Multiplication	61.31	52.40	254.68	19.32
GeoMean	**65.81**	**50.39**	**239.06**	**19.16**

time taken to JIT-compile the Java bytecodes to OpenCL code. In contrast, the driver compilation times are the reported times for compiling the OpenCL code to machine code.

To better understand the JIT-compilation overheads, the *Matrix Multiplication* benchmark was further analysed since it combines both the local memory allocation and loop tiling. The JIT compilation of that benchmark takes up to 63.7% of more time than the original TornadoVM JIT compiler. From that additional compilation time, the newly introduced optimisation phases account for up to 25%. The rest of the overhead is distributed amongst the rest of the compilation phases, and they are attributed to the increased size of the IR graph. The addition of local memory and loop tiling awareness to the IR graph results in up to 50% additional nodes that are processed by subsequent optimisation. The occurrence of extra nodes processed by the consequent optimisation phases is translated to an approximately 35% increase of compilation time. Nevertheless, the percentage of compilation time in the total execution time is less than 5% and, as in any other optimising JIT-compiler, this overhead is encountered only once during execution (the initial compilation).

5.6 Summary

The work presented in this chapter enables an alternative approach to exploit local memory allocation automatically and transparently with Just-In-Time (JIT) compilation. A set of compiler extensions that allow arbitrary Java programs to utilise local memory on GPUs without explicit programming has been implemented in TornadoVM and its JIT compiler. A selection of benchmarks and different GPU architectures used to

evaluate the performance gains and GPU vendor portability. The results showcased performance speedups of up to 2.5x compared to equivalent baseline implementations that do not utilise local memory or data locality. Furthermore, the generated code was compared against hand-written optimised OpenCL code to assess the upper bound of performance improvements that can be transparently achieved by JIT compilation without trading programmability. The results highlight that the complete optimisation process can reach the efficiency of the generated code.

The following chapter aims to break the boundaries of single device performance, and enable concurrent execution on multiple heterogeneous devices. It explains how a heterogeneous managed runtime can enable multiple Java methods to be deployed on multiple heterogeneous devices concurrently, while leveraging a Machine-Learning model for an intelligent task-to-device allocation.

Chapter 6

Intelligent Scheduling of Multiple-Tasks on Multiple-Devices (MTMD)

The previous two chapters addressed the two distinct challenges of bridging the gap between heterogeneous hardware and managed runtime systems. Chapter 4 addressed the issue of enabling seamless integration of FPGAs into managed runtimes while Chapter 5 improved performance of the auto-generated GPU code for Java programs. This chapter extends beyond single device performance by presenting a mechanism to allow multiple tasks deployed by a heterogeneous managed runtime to run onto multiple devices along with intelligent device/task allocation.

To ease the transition towards heterogeneous parallel programming models, research has focused on making high-level programming abstractions widely available [KDA+20]. For instance, TVM [CMJ+18] is a flexible machine learning compiler framework for CPUs, GPUs, and machine learning accelerators. Moreover, Halide [AMA+19] is a programming language for image processing pipelines on CPUs, GPUs, and FPGAs. In addition, approaches like IBM J9 [IHKS15] with GPU support, StreamIT [UGT09, HHWG12], Aparapi [AMD16] and TornadoVM [FPZ+19] allow Java programs to be executed on heterogeneous hardware. However, although the solutions mentioned above aim at closing the programmability gap, they tend to focus on the execution and utilisation of a single device. Since the availability of multiple devices within a computing platform has become the new norm, high-level programming frameworks also need to schedule, orchestrate, and scale up the executed programs on a large number of diverse hardware without depending on the user's expertise.

This chapter presents a novel Multiple-Tasks on Multiple-Devices (MTMD) mechanism that allows seamless concurrent heterogeneous execution of Java programs. Such functionality is achieved by a system that uses and extends the virtualisation layer of TornadoVM. The key novelty is the decomposition of the input applications at task-level granularity. To achieve this decomposition, the original TornadoVM system was augmented with subsystems to automatically perform data dependency analysis and generate a set of blocks of bytecodes for enabling concurrent execution on heterogeneous devices. Each available device is assigned a Java-thread that runs an instance of the interpreter executing the generated bytecodes (discussed in Section 2.3.1).

Since concurrency does not implicitly guarantees high-performance without the efficient allocation of tasks to devices, a Machine Learning (ML) based scheduling approach was employed. The aim of this model is to allow the dynamic selection of the most suitable device for each task. To achieve that, program features are extracted from the compiler graph and passed onto a pre-trained multiple classifier system that selects the target device among CPUs, integrated GPUs, and discrete GPUs. The combination of parallel bytecode execution, concurrent deployment of execution contexts at task-level granularity, and intelligent mapping of tasks onto the available devices results in the seamless and concurrent execution.

Briefly, this chapter makes the following contributions:

- A novel mechanism for enabling Multiple-Tasks on Multiple-Devices (MTMD) execution for Java programs by utilising the available OpenCL-compatible devices existing on the host system.
- A static code feature extractor from the Graal compiler IR for training a Machine Learning model for devices of specific compute capabilities.
- A multiple-classifier system capable of allocating tasks onto a suitable device selected among the available CPUs, integrated GPUs, and discrete GPUs.
- A detailed evaluation of the proposed approach across twelve applications. Evaluation of the concurrent execution showcased up to 83% performance improvement against the highest-performing single device, and up to 91% of the performance of the theoretical best allocation and concurrent execution of tasks to heterogeneous devices.

The rest of this chapter is organised as follows: Section 6.1 describes the motivation and provides the required background information, Section 6.2 outlines the architecture

of the MTMD mechanism with its key components, Section 6.3 describes the ML architecture as well as the scheduling policies for the task-device selection. Finally, Section 6.4 presents the experimental setup and performance evaluation of the MTMD.

6.1 Motivation: Beyond Single Device Performance

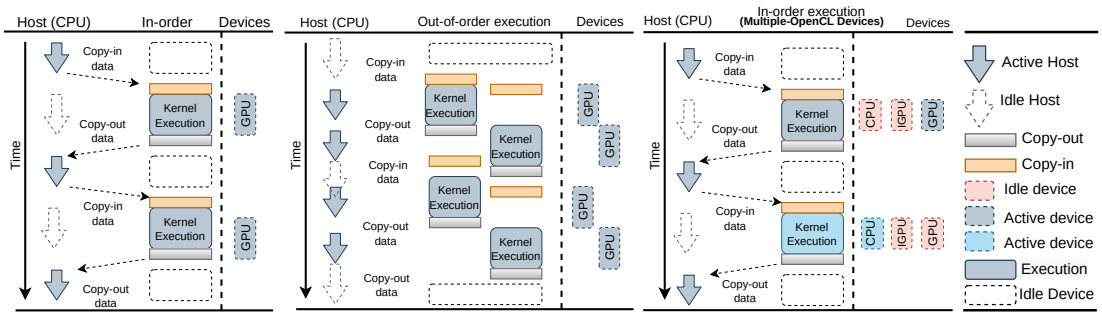
6.1.1 An OpenCL Review on Multiple Devices

As discussed in Section 2.1.2.1, OpenCL [SGS10] is one of the first standards for heterogeneous programming. It offers a uniform Application Programming Interface (API) and a device platform abstraction that allows all different types of devices to be programmed in the same portable way. By using OpenCL, developers can enable their programs to exploit hardware accelerators through task and instruction-level parallelism.

Throughout the years, the OpenCL standard has been extended to optimise resource utilisation of modern heterogeneous hardware. One aspect of the OpenCL extensions is the introduction of different execution modes for single and multiple device configurations. Figure 6.1 exemplifies the three currently supported execution modes of OpenCL: a) in-order single-device execution, b) out-of-order singe-device execution, and c) in-order multiple-devices execution.

When utilising in-order single-device execution [MO16, MO14], as shown in Figure 6.1(a), developers can offload parts of their programs for acceleration on a single OpenCL-compatible device. Also, in this mode, data copying between the host and the device never overlaps with the kernel execution on the device. This results in a strictly sequential in-order execution mode in which the device can remain idle between the intervals of data copying and execution. To mitigate the introduction of idle cycles, OpenCL introduced the out-of-order execution mode (Figure 6.1(b)) in which developers can overlap data copying and kernel execution. Although a single device is still utilised in this mode, the idle cycles are reduced by simultaneously copying data between the host and device while executing code on the accelerator. Finally, the last execution mode of OpenCL regards the multi-devices execution, as shown in Figure 6.1(c). In this mode, developers can build multiple contexts (one per device) and utilise more than one accelerator from within their programs. However, this mode relies on the user to define the execution queues, as well as to encode every aspect.

To address the limitations and the idle cycles introduced by the multi-devices in-order execution mode of OpenCL, several frameworks have been proposed to facilitate



(a) In-order execution. (b) Out-of-order execution. (c) Multi-devices execution.

Figure 6.1: Overview of OpenCL execution modes (Out-of-order on Single Device vs In-order on Multiple Devices).

programming on multiple devices. For instance, VirtCL [YWTC15], SnuCL [KSL+12], PySchedCL [GSK+20], FluidiCL [PG14], MultiCL [APBcF16], EngineCL [NBB19] and SOCL [HBD+13] focus on single or multi-task level scheduling for standalone or partitioned OpenCL applications. A common denominator of all these frameworks is that they solely emphasise on non-managed applications, thereby leaving the area of managed languages unexplored. Exploiting multi-device concurrency and scalability via managed programming languages poses significant challenges due to the need for multi-level compilation. Therefore, several research directions are available due to the dynamic nature of managed languages and heterogeneous platforms.

6.1.2 The TornadoVM Perspective

As presented in Section 2.3.1 TornadoVM allows developers to compose groups of multiple tasks (called `TaskSchedules`) that can execute on a hardware accelerator. However, these `TaskSchedules` can only target a single device without allowing different tasks within a task-schedule to be executed concurrently on various accelerators.

To understand the performance implications of these limitation, a blur filter application was implemented and evaluated, while using the TornadoVM framework. Listing 6.1 outlines the application that consists of three compute kernels, each operating independently on an `RGB` pixel of the input image. This listing showcases three different tasks that operate on different instances of the same method, but on different data. Each of the tasks invokes an instance of the `compute` method with a read-only instance of the image object. Moreover, each of these tasks calculates the results of the altered pixels and stores them into a separate primitive array. Therefore, this workload provides an interesting case of multiple methods operating on different data that can

execute in parallel.

The experimental setup (described in detail in Section 6.4) to evaluate the blur filter application is a commodity laptop. This machine is equipped with three OpenCL-compatible devices: 1) a multi-core CPU (Intel Core i7-9750H), 2) an integrated GPU (Intel UHD Graphics 630), and 3) a discrete GPU (NVIDIA GeForce GTX 1650). The underlying idea of using a personal computer is to highlight that Java developers and applications can benefit if they could use heterogeneous concurrency from this type of setups.

Since in its current state TornadoVM can only schedule all tasks to execute on a single device, an optimisation opportunity is missed since the available device are under-utilised due to the lack of support for concurrent execution. Figure 6.2 depicts the evaluation results from running the blur filter with two data sizes (1K and 4K images) across the three different devices: 1) running all tasks on the CPU, 2) running all tasks on the integrated GPU, and 3) running all tasks on the discrete GPU.

As shown in Figure 6.2, running all tasks on the discrete GPU yields the best performance for the blur filter application by up to 3.15x compared to a serial Java implementation. However, since the tasks are executed in-order, both the integrated GPU and the CPU remain idle without exploiting the potential performance through multi-device execution. To enable concurrent execution by allowing tasks within a `TaskSchedule` to execute on different devices simultaneously, a novel *Multiple-Task Multiple-Device (MTMD)* execution mode has been introduced in TornadoVM.

6.2 Multiple-Tasks on Multiple-Devices

Several key components have been modified or introduced to enable the *Multiple Tasks Multiple Devices (MTMD)* execution mode in TornadoVM. Figure 6.3 outlines both the original TornadoVM software stack (at the top), as well as the required modifications for

Listing 6.1: Example of the TornadoVM Task-based Parallel API with multiple Tasks.

```
1 TaskSchedule filter = new TaskSchedule("blur")
2  .task("red", BlurFilter::compute, redFilter, image)
3  .task("green", BlurFilter::compute, greenFilter, image)
4  .task("blue", BlurFilter::compute, blueFilter, image)
5  .streamOut(redFilter, greenFilter, blueFilter)
6  .execute()
```

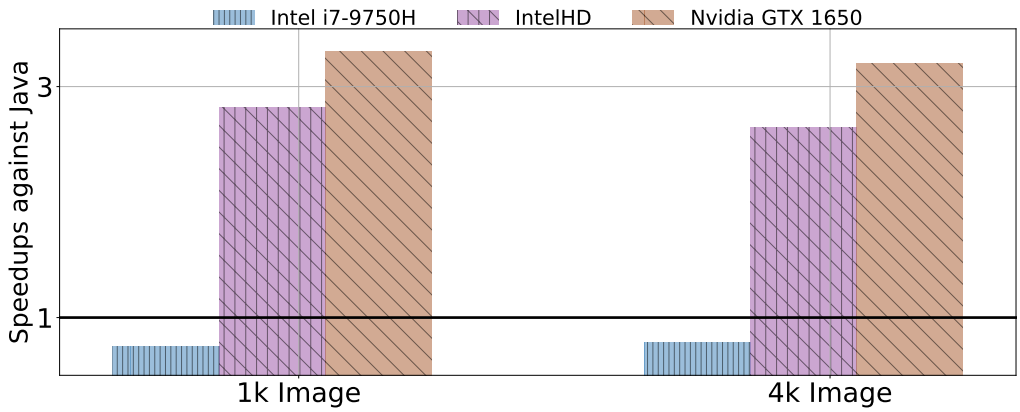


Figure 6.2: Attainable performance speedups against sequential Java for a CPU, an integrated GPU and a discrete GPU.

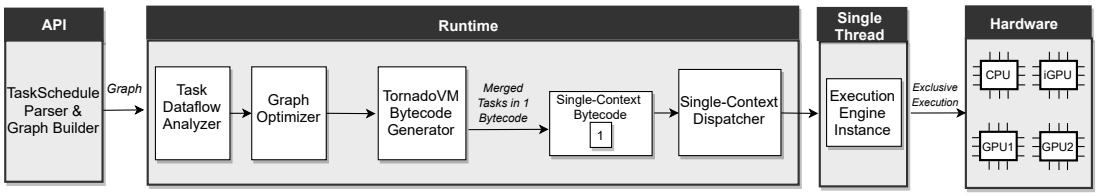
enabling MTMD execution (bottom). As shown in Figure 6.3(a), TornadoVM utilises its own API (described in Section 2.3.1) to create `TaskSchedules` which are then parsed to create dataflow graphs that contain the various input tasks. These graphs are then analysed and optimised during runtime, and, in turn, several TornadoVM-specific bytecodes are generated (presented in Table 2.2 of Section 2.3.1). In the original TornadoVM, all bytecodes that correspond to all tasks of a particular `TaskSchedule` are enqueued into a single context buffer and are consequently dispatched for execution by a single instance of the execution engine. Therefore, all bytecodes, and consequently, all tasks of a `TaskSchedule` can only run on a single device at a time.

As shown in Figure 6.3(b), to enable concurrent execution in TornadoVM, several components have been modified (light blue) or introduced (dark blue):

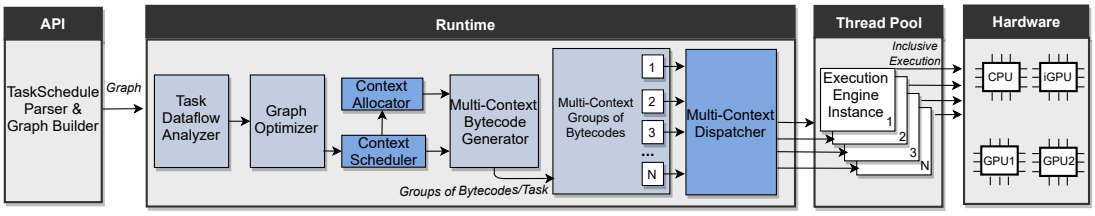
1. The `Task Dataflow Analyser` and `Graph Optimiser` components are responsible for analysing the dependencies between tasks and optimising the graph, before scheduling them onto the devices. Both have been modified to enable concurrent execution by detecting tasks that do not share dependencies among them.

2. The `Context Allocator` component that creates groups of dependent tasks has been introduced. This module ensures that are scheduled together.

3. The `Context Scheduler` component that schedules dependent task groups onto devices has also been introduced. The prior implementation only "locked" a single device, thus all tasks were scheduled there.



(a) The original TornadoVM software stack.



(b) The concurrent MTMD TornadoVM software stack.

Figure 6.3: High-level overview of the components added and modified to the original TornadoVM to enable the concurrent MTMD execution.

4. The `Multi-Context Bytecode Generator`, which is an extension of the TornadoVM bytecode generator, is responsible for generating bytecodes for multiple target devices concurrently instead of a single one.

5. The `Multi-Context Dispatcher` has been introduced to assign bytecodes, that belong to a task group, to a particular execution engine instance. The execution instances are implemented as a Java-thread-pool of execution engines that run the TornadoVM interpreter; with each one being responsible for executing a single context on a single device.

6.2.1 Task Dataflow Analyser

As shown in the example of Listing 6.1, a `TaskSchedule` in TornadoVM can be composed of multiple tasks that may have data dependencies between them, i.e., the output of one task can be the input to another. Since developers can compose arbitrary `TaskSchedules`, the presence or the absence of dependencies between tasks is not guaranteed. To enable concurrent execution of arbitrary tasks on different devices, the modification of the Task Dataflow analyser and the Graph optimiser to extract inter-task dependencies was necessary.

While analysing the tasks of a given `TaskSchedule`, TornadoVM generates Java bytecodes for each task which are then transformed into a compiler graph based on the

Intermediate Representation (IR) [DWS+13] of the TornadoVM compiler. The dataflow analysis phase has been implemented as a compiler phase in the JIT Compiler. This phase detects the input and output arguments of the original TornadoVM tasks (Java methods). After the dependencies are identified, the task dependency graph is traversed to create a map of their accessibility within the different tasks of a `TaskSchedule`. Then, each input/output argument of each task is marked as `READ`, `WRITE` or `READ_WRITE` and stored as task meta-data information. This process is completed when the last task of the input `TaskSchedule` has been analysed and evaluated correctly.

At the end of the dataflow analysis phase, the captured meta-data is used to create a Direct Acyclic Graph (DAG) of the intra-`TaskSchedule` dependencies. This information is stored at runtime to be later accessible for scheduling dependent tasks on the same device to avoid costly data copying of temporary variables between devices. In contrast, independent tasks are grouped and scheduled independently for concurrent execution across numerous hardware accelerators.

To avoid tasks that are sharing only read-only objects to be grouped, the Graph optimiser phase was extended with an additional optimisation phase. The proposed optimisation tackles `READ`-only dependencies between tasks by duplicating the `READ`-only parameters between them. However, this process also duplicates the variables that lie in the Java-heap on the host side, and increasing the memory footprint of the complete application. In this way, tasks become independent and can be executed concurrently.

6.2.2 Context Allocator and Scheduler

Based on the task meta-data derived from the dataflow analysis and optimisation phases, tasks can be grouped or remain independent. Each group consisting of one or more tasks will be then be assigned to a device for execution via a device `context`. The context defines an independent computational entity (a single task or a multiple tasks) that can target a single device. As soon as contexts are defined, they also lock the allocated devices.

At this point, the scheduling of tasks on devices happens statically without considering specific task characteristics, such as memory accesses, parallel dimensions, and single or double-precision operations. Tasks are assigned onto the available devices in a *First Come First Served* order, and they are parsed in the order they are attached on the *TaskSchedule*. Also, devices are ordered based on their characteristics and computational capabilities. Section 6.3.4 discusses in-depth how this scheduling approach was

Listing 6.2: Example of TaskSchedule with multiple independent tasks.

```
TaskSchedule graph = new TaskSchedule("workload")
  .task("t0",DFT::dft, inReal,inImag,outReal,outImag)
  .task("t1",Blackscholes::bs,input,callPrice,putPrice)
  .task("t2",MM::mm, matrixA, matrixB, matrixC, mmSize)
  .streamOut(outReal,outImag,callPrice,putPrice,matrixC)
  .execute();
```

augmented by introducing predictive modelling based on features captured from any given method.

6.2.3 Multi-Context Bytecode Generator

The previous steps reduced the computational granularity of a `TaskSchedule` to multiple contexts consisting of single or multiple inter-dependent tasks. At this point of the execution, TornadoVM creates internal TornadoVM-specific bytecodes [FPZ+19] that orchestrate the execution, the synchronisation, and the data exchanges between the host and devices. The purpose of this additional virtualisation layer is to abstract away from the developers all the mechanics and details of hardware acceleration and kernel offloading to any device. In the original TornadoVM, since tasks within a `TaskSchedule` can only be executed on a single device, the bytecode generator creates single context bytecodes which are destined to be executed in-order on a particular device.

To enable concurrent execution onto the available devices, the existing virtualisation layer was extended to embed device selection control at the task-level (rather than in the original `TaskSchedule`-level).

Listing 6.2 showcases three applications grouped as independent tasks in the same `TaskSchedule`. These tasks are DFT, BlackScholes and Matrix Multiplication (MM). Initially, the dependency analysis marked them as independent and during context allocation with First-Come First-Served (FCFS) scheduling, all tasks were assigned to the available devices.

Since tasks are independent, the introduced multi-context bytecode generator creates three independent sets of bytecodes. Listings 6.3, 6.4, and 6.5 correspond to the generated multi-context bytecodes for tasks `t0`, `t1`, and `t2`, respectively.

The bytecodes of each context are assigned to a separate device (if three are present) awaiting interpretation and execution by TornadoVM.

Listing 6.3: TornadoVM bytecodes for task: t0 (DFT).

```
BEGIN    <0>    //New context [device 0]
COPY_IN <0, bi1, in>      //Copies <in>
COPY_IN <0, bi2, in>      //Copies <in>
COPY_IN <0, bi3, in>      //Copies <in>
COPY_IN <0, bi4, in>      //Copies <in>
LAUNCH  <0, bi5, @dft, in, temp>
COPY_OUT_BLOCK <0, bi6,out>    //C-out
COPY_OUT_BLOCK <0, bi7,out>    //C-out
END      <0>              //Ends context
```

Listing 6.4: TornadoVM bytecodes for task: t1 (BlackScholes).

```
BEGIN    <1>      //New context[device 1]
COPY_IN <1, bi1, in>       //Copies <in>
ALLOC   <1, bi2, out> //Allocates <out>
ALLOC   <1, bi3, out> //Allocates <out>
LAUNCH  <1, bi4, @bs, temp, out>
COPY_OUT_BLOCK <1, bi5,out>     //C-out
COPY_OUT_BLOCK <1, bi6,out>     //C-out
END      <1>               //Ends context
```

Listing 6.5: TornadoVM bytecodes for task: t2 (Matrix Multiplication).

```
BEGIN    <2>//New context [device 2]
COPY_IN <2, bi1, in>  //Copies <in>
COPY_IN <2, bi2, in>  //Copies <in>
COPY_IN <2, bi3, in>  //Copies <in>
LAUNCH  <2, bi4, @mm, in, temp>
COPY_OUT_BLOCK <2, bi5,out> //C-out
END      <2>           //Ends context
```

6.2.4 Thread Pool of Execution Engines

To execute the multi-context bytecodes in parallel, a scalable thread-pool of execution engines was implemented. Each of the execution engines is responsible for interpreting the bytecodes corresponding to a context assigned to a specific device, as shown in Figure 6.3(b). These bytecodes can correspond up to several tasks with or without dependencies among them.

Each of the execution engines deploys an isolated instance of the interpreter per device that executes the multi-context bytecodes assigned to it. Following the original TornadoVM execution flow, tasks can be dynamically compiled to OpenCL, and the

execution engines can access binaries from a global code cache. The interpreter can be JIT compiled by the underlying JVM (e.g., Oracle HotSpot) to improve performance. The exact architecture has been previously explained in Section 2.2.2. Note that the TornadoVM bytecodes only orchestrate the execution between the accelerators and the host machine and do not perform the actual computation [FPZ+19]. The latter is achieved by executing the generated OpenCL code via the device driver.

Another benefit of reducing the granularity of the execution from a `TaskSchedule`, to smaller groups of tasks composing a context, is the ability to increase the resiliency of the execution by enabling fault tolerance, which in turn reduces the cost of re-execution.

6.2.5 Discussion

The components described in the previous subsections, have been added or augmented to enable concurrent execution of fine-grained tasks on multiple accelerators. The performance benefits of enabling concurrent execution across devices within the same computing system were assessed. To do so, the results obtained through the blur filter application of Listing 6.1 were revised. In the revised experiments, the concurrent execution of the original independent tasks is now enabled, while using the default scheduling policy. Figure 6.4 adds three additional data points to Figure 6.2 which correspond to three additional execution scenarios:

(a) **In-order-execution** of all tasks on the CPU, integrated GPU (IGPU), and discrete GPU (grey bar).

(b) **Concurrent-execution** of all tasks across all devices (first running on the CPU, second on the IGPU, and third on the discrete GPU - orange bar).

(c) **Concurrent-execution** of all tasks across two devices (first two on the discrete GPU, and third on the IGPU - red bar).

As shown in Figure 6.4, the additional execution scenarios can influence dramatically the performance, which can be up to 2× higher compared to running the whole `TaskSchedule` on the same device. However, the problem of statically deciding which policy to employ, for scheduling fine-grained tasks across the available accelerators, is particularly challenging due to the diverse characteristics and performance of each task. To enable efficient scheduling that considers device availability, the potential performance benefits of concurrent execution, and code characteristics, an ML-based

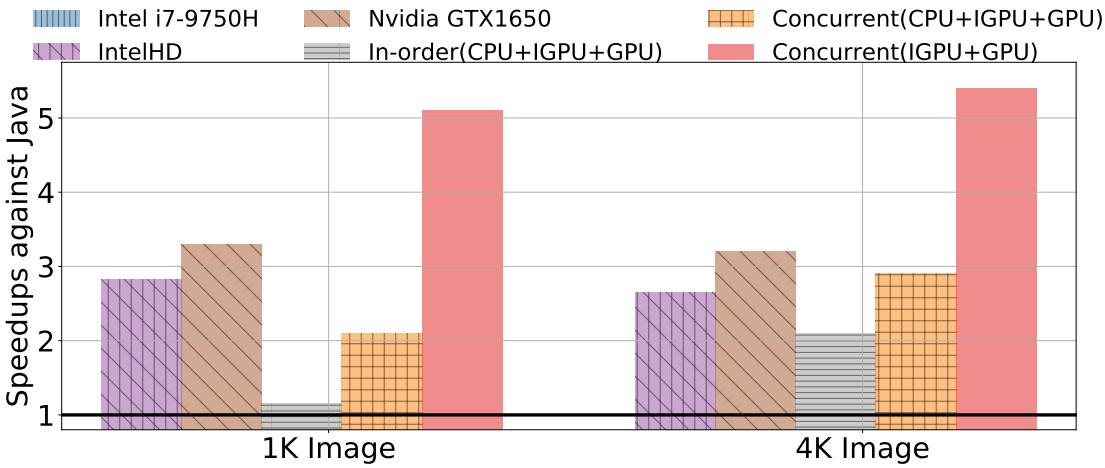


Figure 6.4: Concurrency limits: Achieved speedups against sequential Java for a CPU, an integrated GPU, Discrete GPU, in-order on multiple devices, concurrent into three devices and concurrent into two devices.

scheduling technique was introduced. The following section presents the design, modelling and implementation of a custom ML architecture that fulfills the requirements of the system.

6.3 Prediction-based Scheduling for MTMD

Section 6.2 outlined the minimum required runtime support for a heterogeneous managed runtime to efficiently handle the orchestration of dispatching multiple tasks on multiple devices concurrently. However, to fully utilise a system with these capabilities, being able to perform an efficient task/device allocation is required. To that end, a Machine Learning (ML) model, trained to perform the task of device-task allocation, governs the underlying scheduling policy.

A decisive factor for finding a suitable scheduling strategy is to detect the best computing device for a given task in terms of attainable performance. This work focuses on commodity personal computers due to the broad set of heterogeneous hardware available. However, it can be extended to other computing systems equipped with a larger number of hardware devices. The experimental platform contains a CPU, an Integrated GPU, and a Discrete GPU. To train the ML model, a set of features describing the application is extracted from the compiler IR (Graal IR [DWS+13]) before generating the OpenCL kernel for a given task. The specifics of the Graal IR were presented in Section 2.2.3.1.

Consequently, the next step is to use a system to determine the optimal mapping. Based on the insights presented in prior-work [GWO13, WGO15] tackling the same problem from a pure OpenCL perspective, a solution of this mapping can be a Multiple-Classifier-System (MCS). Each component of this system is a tree-based two-class Binary classifier, trained to compute the probability that a specific task will exhibit speed-up when executed on one device over another. The final decision is made through the conjunction of the output probabilities of the learners mentioned above. The following subsections describe in detail the components of the proposed ML-based scheduling policy to enable efficient execution of the MTMD model.

6.3.1 Feature Extraction

Extracting meaningful characteristics from an input application is the decisive factor for effectively predicting which task will perform better across different devices. Prior work, discussed in detail in Section 3.4.2, proposed several methodologies for extracting code features directly from OpenCL kernels. Such approaches are not completely aligned with this work due to the two-stage compilation process that TornadoVM employs (from Java to OpenCL C, and from OpenCL C to binary code). In contrast, as the given experimental platform performs device-specific optimisations during code generation, in this work, it is important to be able to obtain meaningful insight from the application level instead of the generated code. Other approaches rely on extracting features from unmanaged languages, while in this context, applications are written in Java. Also, the given applications do not follow traditional parallel programming patterns. Hence, this work relies on a feature extraction process from the compiler IR graph of the initial Java method during JIT compilation. Following this approach, sufficient information is captured for corelating the behaviour of the auto-generated OpenCL executable and the input program written in Java.

When a task is assigned to a `TaskSchedule`, the graph builder constructs the corresponding bytecodes to an IR graph. Then, the TornadoVM JIT compiler starts optimising the graph from a high-level to a low-level. A decision was made to make decision from an IR representation closer to the original bytecodes of the program, thus features are extracted at the end of the sketch-tier compilation. This is achieved by adding a `Feature extraction phase` to obtain the number and type of operations (e.g., loop bounds for calculating the parallel dimensions) based on individual nodes. In practice, this sketch-tier phase also used to populate meta-information obtained directly from an early unoptimised compiler graph for a given method. Therefore, the design

Table 6.1: List of raw features captured from early stage of the compilation from the IR graph.

Type	Feature
Memory	Global Memory Reads (G/R)
	Global Memory Writes (G/W)
	Constant Memory Reads (L/R)
	Constant Memory Writes (L/W)
	Local Memory Reads (L/R)
	Local Memory Writes (L/W)
	Private Memory Reads (L/R)
	Private Memory Writes (L/W)
Loops	Parallel Loops
	Total Loops
Control Flow	Branches
	Switch Cases
	Float Comparison
	Integer Comparison
	If Statements
	Local/Global Barriers
Operations	Boolean Operations
	Cast Operations
	Integer & Float Operations
	Vector Operations
Optimisations	Vector Types
	Pragmas

choice of obtaining features at this point (before code generation) adds modularity to the original system since it can cater other backends or pure x86 execution through Java. Note, this approach developed on top of the version presented in Chapter 5, thus nodes regarding memory accesses are captured effectively.

In the Graal IR graph, several nodes contribute to the structure of the same operation. Hence, during the traversal of the graph, different node patterns capture different operations. For instance, `StoreIndexedNodes` are associated with storing into arrays. However, to make a distinction between stores to global or to the local memory one needs to identify the origin node of the `AddressNode`. Table 6.1 outlines the complete selection of operations captured by traversing the compiler graph. These features are later combined with runtime information regarding the input/output data sizes, number of threads (i.e., work-items) to be deployed, and inter-task dependencies.

For example, Listing 6.6 shows the generated `JSON` file containing all the static

```
"nBody": {
    "DEVICE_ID": "0:2",
    "DEVICE": "GeForce GTX 1650",
    "Global Memory Loads":  "15",
    "Global Memory Stores":  "6",
    "Constant Memory Loads":  "0",
    "Constant Memory Stores":  "0",
    "Local Memory Loads":  "0",
    "Local Memory Stores":  "0",
    "Private Memory Loads":  "20",
    "Private Memory Stores":  "20",
    "Total Loops":  "2",
    "Parallel Loops":  "1",
    "If Statements":  "2",
    "Integer Comparison":  "2",
    "Float Comparison":  "0",
    "Switch Statements":  "0",
    "Switch Cases":  "0",
    "Vector Operations":  "0",
    "Integer & Float Operations":  "57",
    "Boolean Operations":  "9",
    "Cast Operations":  "2",
    "Float Math Functions":  "1",
    "Integer Math Functions":  "0"
}
```

Listing 6.6: Example of static feature extractor output in JSON format for the N-Body simulation method.

features that have been extracted from a method that computes the N-Body simulation. This JSON file will be the input for the model during the inference process along with the runtime information. If a *TaskSchedule* consists of multiple tasks, all tasks are composed into a single JSON file.

6.3.2 Feature Selection & Engineering

Figure 6.5 displays a heat map of the features that were initially extracted from the compiler graph and the runtime. The heat map provides a visual representation of the correlation between the various features. The initial extraction led to 26 distinct features which have been pre-processed and combined to construct new features. These combined features have greater predictive ability compared to the initial ones. During this process, the feature set is further expanded to also include interaction features,

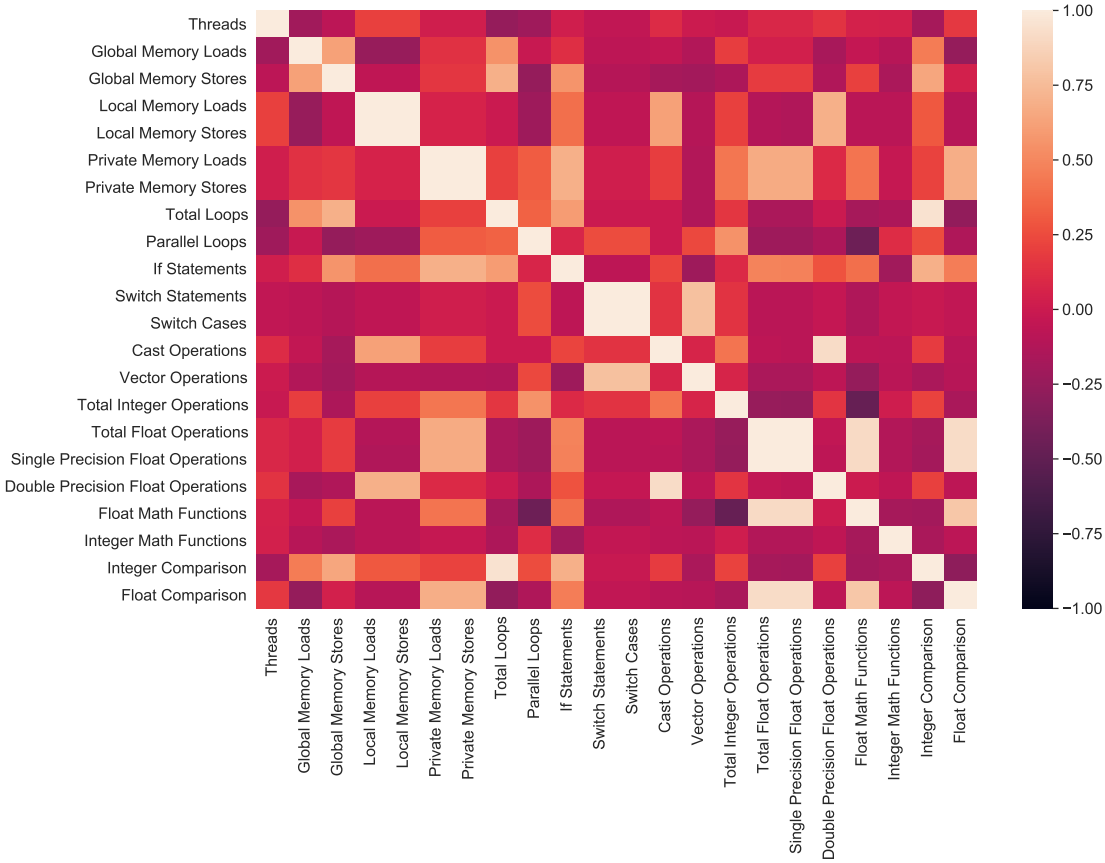


Figure 6.5: A heat map of all the feature variables along with their relation to the target variable (i.e, device speedup). The scale highlights with light color the highly correlated features and with dark color the least correlated features.

i.e., features that are computed as the pairwise product of the existing ones. Furthermore, features that are most relevant to each other (e.g., float_math_function, integer_math_function) are grouped together.

Upon completion of the feature engineering process, the data from features is increased considerably. In such cases, only features that are key attributes of the model are selected. Therefore, the learning algorithm (discussed in Section 6.3.4) focuses only on the most important variables. Also, it prevents the modelling of any underlying noise in the data which may be induced by irrelevant features.

As the underlying approach is tree-based, one of the measures to quantify the importance of the features for the model is the Gini importance [Far10]. This approach calculates the importance of each feature as the sum over the number of splits (across all trees). Based on this metric, the ten features that have the greater impact on the outcome of each classifier are depicted in the Hinton diagram [BGD94] of Figure 6.6.

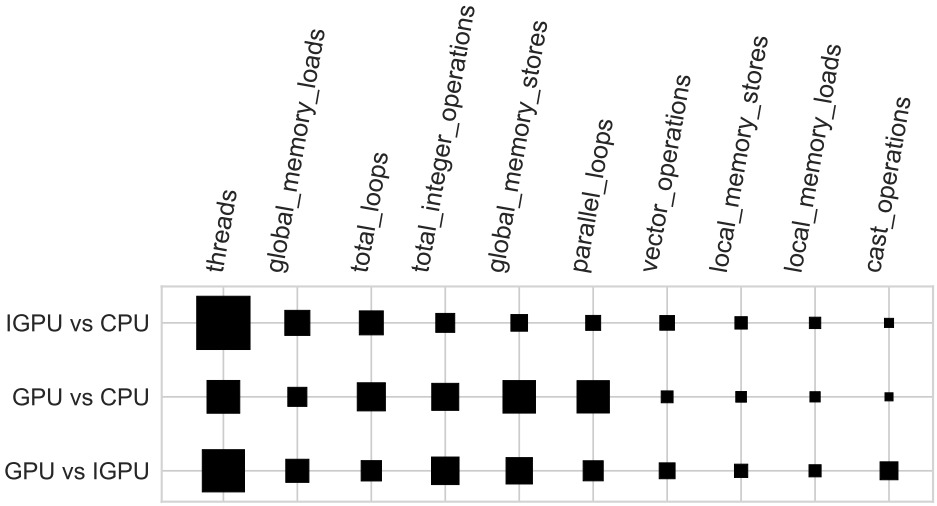


Figure 6.6: Feature importance for classifiers: 1) IGPU vs GPU, 2) GPU vs CPU and 3) GPU vs IGPU. Squares are representing the impact of the feature in the final decision (larger squares have more influence).

The sizes of the squares represent the magnitude of the values, i.e., the corresponding Gini importance of each feature.

6.3.3 Training Dataset

The dataset consists of the static code features extracted from a series of Java applications with different compute characteristics. Each of the applications was executed with various input sizes. Also, each entry of the dataset contains the execution times, while running through TornadoVM on the three available devices, i.e., CPU, IGPU, GPU. Based on these timings the following speedup ratios are computed:

$$\frac{IGPU_{\text{execution time}}}{CPU_{\text{execution time}}} \tag{6.1}$$

$$\frac{GPU_{\text{execution time}}}{CPU_{\text{execution time}}} \tag{6.2}$$

$$\frac{GPU_{\text{execution time}}}{IGPU_{\text{execution time}}} \tag{6.3}$$

These equations represent the binary target variables indicating whether the specific task has speedup on a given device. More specifically, ratios lower than 1.0 indicate a slowdown, and so they are mapped to 0, while ratios above the same threshold correspond to speed up and consequently are mapped to 1. Each of these binary

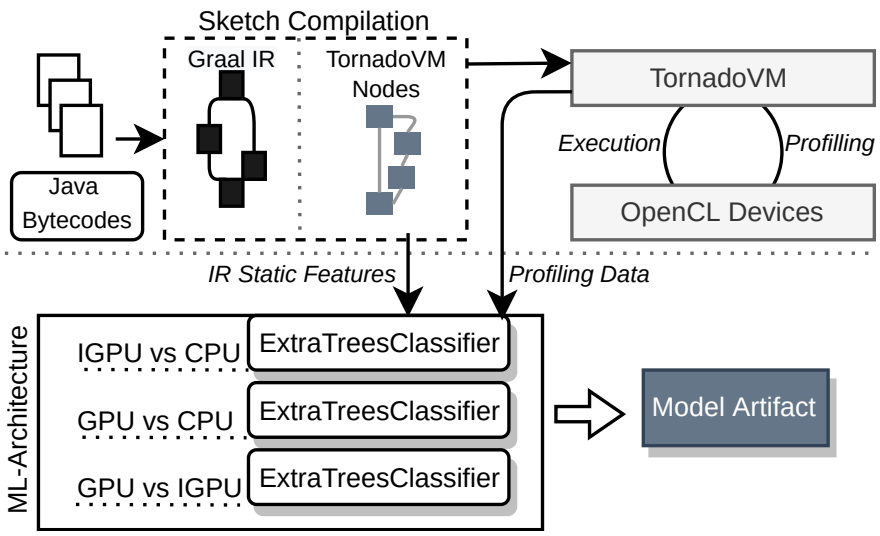


Figure 6.7: An overview of the offline training process of Java programs supported by TornadoVM.

variables will serve as the target for a classifier in the complete multiple-classifier system.

Figure 6.7 showcases the offline process for collecting the data and training the model. Regarding the selection of the input applications, a decision was made to use kernels from the benchmark suite and examples that already exist in the TornadoVM repository. The system relies on a feature extraction process through the IR (generated from the original Java methods). Therefore, the model is purely trained with Java benchmarks compatible with TornadoVM.

Using hand-tuned OpenCL programs can also be an option; however, they capture different performance and programming patterns compared to the OpenCL automatically generated from Java. Previously, Section 5.5.2.2 highlighted that pure OpenCL and auto-generated through TornadoVM have a performance gap. Therefore, extending the training set with benchmark suites purely written in OpenCL, such as Rodinia [CBM+09], OpenDwarfs [KFAB16], Spector [GAMK16b] or PolyBench [GGXS+12] will negatively influence or bias the accuracy of the predictor. This negative influence will be the result of training the model with handwritten and hand-optimised OpenCL programs, while the system is expected to be used with auto-generated OpenCL.

The selected programs were executed with various input configurations, depending on their computational intensity, on an Intel CPU, an Intel HD Graphics, and an Nvidia GTX 1650. For each data point (i.e., input size, and timing counters), the existing profiling infrastructure is used. In more detail, all profiling-events at the OpenCL side

and runtime dynamic information overheads present in the Java side are obtained from the TornadoVM profiler. Overall, the ML model used more than 200 individual data points for training. All data points contain the following entries: method name, input size, extracted features, speedup on CPU, speedup on IGPU, and speedup on GPU. Thus, for each input Java application, multiple entries per input size are obtained. In this way, the model can capture applications that showcase difference performance behaviour on different devices and for various input sizes.

6.3.4 Machine Learning Architecture

The proposed ML architecture is a multiple classifier system consisting of three classifiers, each targeting one of the available devices. The selected algorithm for these classifiers is tree-based and, more specifically, is the Extra Trees [GEW06]. Extra Trees is an ensemble method that uses Bootstrap Aggregating (Bagging) [OM99]. As an ensemble method, an Extra Trees classifier combines the decisions of multiple decision trees. This approach combined multiple learners to lead into an improved predictive performance compared to individual learners. Each of the grown decision trees is trained using a different subset of the training data and the available feature set; a process called Bootstrap Aggregating. Bootstrap Aggregating is a method specifically designed to tackle over-fitting by reducing the variance in the predictions (these aspects of ML modelling were discussed in Section 2.4). This is particularly important in the context of the MTMD model, where the available dataset is limited and cannot be easily extended. Two key factors contributed to a limited-sized training set; 1) only Java-based, compatible with TornadoVM, benchmarks used, and 2) this work focuses only on a system consisting of three devices. Moreover, the main advantage of the Extra Trees method is that the training can be parallelised since each decision tree can be trained independently. Hence, their individual decisions are combined only at the end of the execution.

Training Model: The training model uses three Extremely Randomised Trees (ExtraTrees) classifiers. Each classifier produces a speedup probability for each task between the following pairs: IGPU\CPU (1st classifier), GPU\CPU (2nd classifier), and GPU\IGPU (3d classifier). Among the available tree-based algorithms, such as Decision Trees [Qui86], Random Forest [Ho95], and Extremely Randomised Trees, the latter was selected due to its ability to better handle over-fitting. The hyper-parameters of the model (e.g., estimators, maximum depth) were optimised by searching over a grid of trials and the combination that yielded the best cross validation score (10-fold) was

retained. Moreover, by investigating the training dataset for each classifier, it was found that the datasets of the first and the second classifiers were highly imbalanced. In more detail, the target classes were unequally represented and thus the models would ignore, and in turn, underperform on the minority class. To tackle this issue, the SMOTE algorithm [CBHK02] was used which upsamples the minority class by synthesising new examples.

1st Classifier: The selected ExtraTrees classifier, i.e., the one that yielded the best cross-validation score, fits 100 estimators with maximum depth set to 50. The first level of prediction considers only the IGPU and the CPU and attempts to determine the most suitable device between them for a given task. The output of the model is the probability at which the given task will have speedup when executed on the IGPU instead of the CPU. By selecting an appropriate threshold, the probabilistic output can then be interpreted as class labels, i.e., IGPU or CPU.

For this selection, the Receiver Operating Curve (ROC) [Bra97] and the Precision-Recall Curve [BEP13] were plotted for various thresholds to understand the trade-off in performance. Given the imbalanced nature of the collected dataset, the F1-score was optimised, i.e., the harmonic mean of precision and recall, instead of accuracy, since the former serves as a better measure for the incorrectly classified cases. For the first classifier, the optimal threshold was determined to be around 0.2 resulting in 0.95 F1-score on the held-out dataset.

2nd Classifier: For the second classifier, the optimal performance was achieved by fitting 500 estimators with maximum depth set to 10. In a similar way, the second classifier is trained to distinguish between tasks based on their relative performance on either the discrete GPU or the CPU. Again, the probabilistic output is turned into a class label, i.e., GPU or CPU. The optimal threshold is determined to be 0.6 with a 0.96 F1-score on the held-out dataset.

3rd Classifier: Lastly, the third ExtraTrees classifier fits 50 estimators while the maximum depth is set to 50. The third classifier aims to select between IGPU and GPU. Following the same process as previously, the best threshold is defined around 0.6 resulting to 0.91 F1-score on the held-out dataset.

6.3.5 On-line Scheduling Process

Figure 6.8 outlines the on-line scheduling process. This process performs the inference by using the pre-trained model. During run-time, the pre-trained ML model is invoked along with a JSON file that contains the features of a task eligible to run on the system.

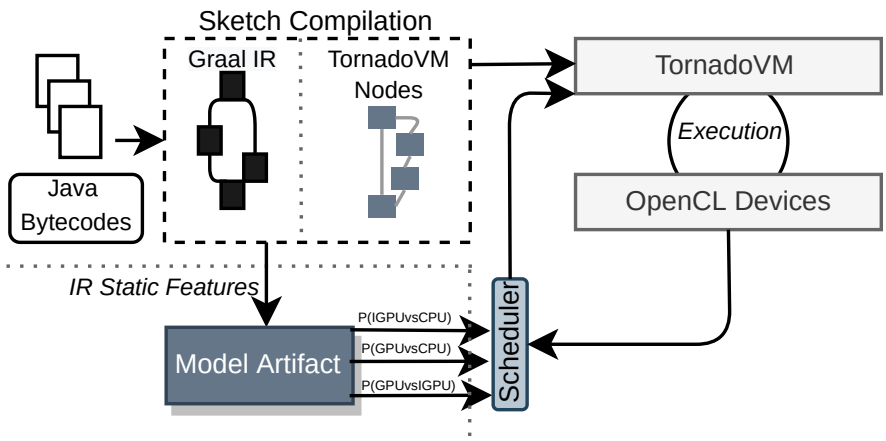


Figure 6.8: Online scheduling based on task-features, available devices and trained model.

Table 6.2: Truth table to perform the final device selection and scheduling.

Classifier			**Target Device**
IGPU vs CPU	GPU vs CPU	GPU vs IGPU	
0	0	0/1	CPU
1	0	0/1	IGPU
0	1	0/1	GPU
1	1	0	IGPU
1	1	1	GPU

These features consist of inputs to the multiple-classifier-system which outputs the three probabilities. By setting the thresholds discussed in Section 6.4.3, the probabilities are converted into class labels, i.e., 0 for slowdown and 1 for speedup. The final decision is taken by using the truth table presented in Table 6.2. More specifically, the following scenarios are considered for each task:

1. **Schedule on CPU:** If a task is predicted to have a slowdown on both IGPU and GPU, then regardless of the verdict of the third classifier, it will be scheduled on the CPU.

2. **Schedule on IGPU:** If a task is predicted to have speedup on the IGPU but slowdown on the GPU, then, regardless of the verdict of the third classifier, it will be scheduled on the IGPU.

3. **Schedule on GPU:** If a task is predicted to have a slowdown on the IGPU but speedup on the GPU then, regardless of the verdict of the third classifier, it will be scheduled on the GPU.

4. **Schedule on GPU or IGPU:** If a task is predicted to have speedup on both the IGPU and the GPU then the decision is being made based on the output of the third classifier. If it is predicted that the task will gain speedup when executed on the GPU versus on the IGPU, then the GPU is selected and vice versa if it is predicted to exhibit slowdown.

6.4 Evaluation

This section presents the experimental evaluation of the proposed MTMD mechanism. Firstly, the experimental setup and methodology are described, as well as the applications used to assess the attainable performance. Finally, this section concludes with a presentation and a discussion of the results of the concurrent device execution and scheduling.

6.4.1 Experimental Setup and Methodology

To assess the performance, a modern commodity computing system was used as an experimental setup. This system was equipped with an Intel CPU, an Intel integrated GPU and a discrete Nvidia GPU. Also, this configuration corresponds to a commodity machine with a high compute capacity, which can be seamlessly utilised by a Java application via the MTMD execution mode. Table 6.3 outlines the hardware and software characteristics of the experimental setup.

Regarding the experimental methodology, this work follows the widely adopted Java evaluation process outlined by Georges *et al*. [GBE07]. Initially, a warm-up phase takes place for every application to stabilise the performance of the JVM. The warm-up phase ensures that the Java code of each application is JIT-compiled, and in this case, 100 iterations was enough to achieve this. Once the warm-up phase is complete, each application runs 10 consequent times and then it reports the mean of the obtained total execution times. These timings also include the time spent for the model inference. Warm-up in this context implies that the metacircular TornadoVM itself is also JIT compiled.

6.4.1.1 Applications and Input sizes

The evaluation of the proposed MTMD mechanism uses twelve applications that can be classified as compute-intensive, memory-intensive, and control-flow intensive. The goal

Table 6.3: Experimental Setup.

Hardware	
Processor	Intel Core i7-9750H CPU @ 2.60GHz
Cores	6 (12 HyperThreads)
RAM	32GB
Integrated-GPU	Intel UHD Graphics 630
Discrete GPU	NVIDIA GeForce GTX 1650 (Turing) 4GB GDDR5, 896 CUDA Cores
Software	
Operating System	Ubuntu 20.04 (Kernel 5.4.0-52-generic)
OpenCL (CPU)	2.1 Device Version
OpenCL (IGPU)	2.1 Device Version
OpenCL (GPU)	1.2 Device Version
CUDA Driver	450.80.02
TornadoVM	v0.7
JVM	OpenJDK 1.8.0_262 with JVMCI
Java Heap	-Xmx22G -Xms22G

of this selection of applications has been to assess MTMD by running all the applications concurrently. However, the inability of TornadoVM to support data transfers, from the host to the various devices, of sizes over 1 GB, led this work to split the total workload of twelve applications into three groups (`Groups 1 to 3`), as shown in Table 6.4. Each group has a randomly assigned number of applications that can be concurrently executed for different input data sizes (`small`, `medium` and `large`). Table 6.5 presents the input data sizes for each application.

Table 6.4: The input applications written in Java with the TornadoVM API and divided into three distinct groups for the evaluation.

Group	Application	Description
1	DFT [GLD+08]	Hierarchical mixed radix FFT algorithms for both power-of-two and non-power-of-two sizes.
	Black-Scholes [GGKSC13]	Option pricing using the Black-Scholes merton process.
	Matrix Multiplication [VD08]	Matrix multiplication on square matrices.
2	NBody [PJH09]	Particle simulations.
	MonteCarlo [RK16, PCV17, Pap16]	Monte Carlo simulation for option pricing models.
	RenderTrack [NBZ+15]	Parallel kernel for image decomposition that contains multiple control flow operations.
	Mandelbrot [HR15]	Iterative function applied in a large set of points.
	Hilbert Matrix [OYIY18]	Dense matrix computation on a square matrix.
3	Matrix Transpose [VD08]	Matrix transpose operation on a square matrix.
	B&W Filter [Ion17]	A filter that converts an RGB image to Grayscale.
	Convolution [AUvAS+12]	A two dimensional process of adding each element of an image to its local neighbors.
	Euler Method [GLC+11]	A first-order numerical procedure for solving ordinary differential equations (ODEs).

Table 6.5: The input data sizes for each application (task) in three different ranges: small, medium, and large.

	DFT	BS	MM	NBody	MC	RT	Mandelbrot	Hilbert	MT	B&W	Conv	Euler
Low	1024	65536	65536	1024	65536	262144	262144	65536	65536	1K img	16384	512
Medium	16384	524288	262144	2048	524288	1048576	1048576	262144	262144	2K img	262144	1024
Large	65536	1048576	1048576	8192	1048576	16777216	4194304	1048576	1048576	4K img	1048576	4096

6.4.1.2 Scheduling Strategies

For full coverage of the evaluation of the MTMD mechanism, six alternative scheduling policies were employed. Namely, these scheduling policies are the following:

1. ***Dynamic Reconfiguration (DynRec) [FPZ+19]:*** This is the official scheduling policy supported by TornadoVM which examines all the viable configurations exhaustively. Thus, tasks must be executed serially on all devices in order to determine the highest performing one. After the exhaustive execution is performed, TornadoVM stores the device and uses it again during further invocations of the same code. However, slight changes to the executed code or input data sizes will trigger the exhaustive execution again.

2. ***First-Come-First-Served (FCFS):*** Tasks are scheduled to run on devices following the order that the TornadoVM system discovers the OpenCL device drivers. Tasks will be allocated to devices based on the order that they arrive and based on the hierarchy that the OpenCL device drivers appeared in the system.

3. ***GPU-Priority (gpuprio):*** Tasks are scheduled to run on devices following a score that ranks the devices based on their compute capabilities. In the current system, the discrete GPU is the one with the highest compute capabilities.

4. ***CPU-Exclusion (cpuex):*** Tasks are scheduled to run on devices (except CPUs) following the order that the TornadoVM system discovers the OpenCL device drivers.

5. ***ML-based MTMD (mtmd-ml):*** Tasks are scheduled and dispatched to run on devices with respect to the proposed ML-based scheduler (discussed in Section 6.3).

6. ***Oracle:*** This scheduling strategy presents the device that offers the best performance. This strategy is obtained by the exhaustive offline exploration of the overall optimisation space.

The *Dynamic Reconfiguration* policy is the only policy that requires all tasks within a TaskSchedule to be executed on a single device due to the *Single-context Dispatcher* in the original TornadoVM system (Figure 6.3(a)). On the contrary, the remaining scheduling policies exploit the MTMD mechanism and can operate concurrently on multiple devices. Additionally, the *Dynamic Reconfiguration* and the *Oracle* scheduling policies are used to define the peak performance for the serial (single context) and

the concurrent (multi-context) executions. Both approaches introduce a significant overhead that makes them unsuitable for real-time execution. Furthermore, the dynamic reconfiguration and the Oracle configurations do not consider the time taken to obtain the best configuration as it is an unrealistic cost for a real-time system. However, these approaches establish a theoretical peak for single and multi-concurrent execution of the workloads that are assessed in this work.

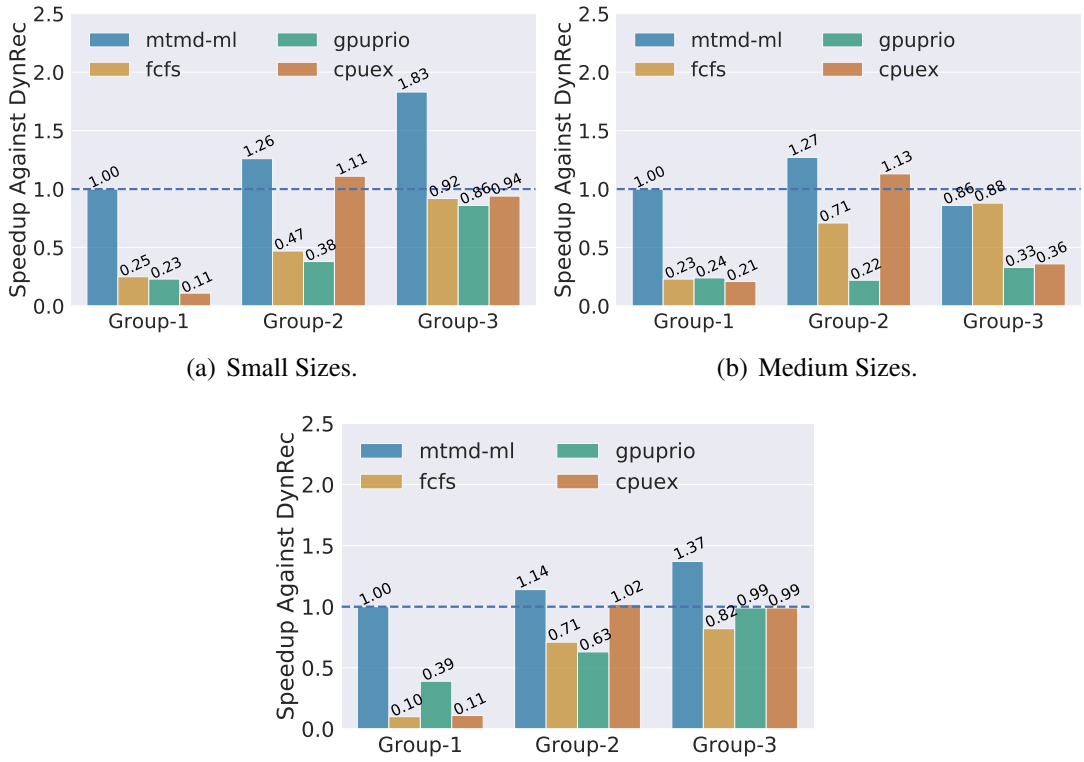
6.4.2 Performance Evaluation of MTMD

This section is seperated into two parts. Section 6.4.2.1 discusses the performance of all scheduling policies that operate with the MTMD execution mode against the best consecutive execution policy, which is *Dynamic Reconfiguration*. Section 6.4.2.2 compares the MTMD scheduling policies against *Oracle*, the best concurrent execution policy.

6.4.2.1 Relative Performance versus Best Consecutive

Figure 6.9 shows the performance comparison of the *fcfs*, *gpuprio*, *cpuex* and *mtmd-ml* policies against *DynRec* for different data sizes (small, medium, large). The *DynRec* policy was used as a reference for the baseline performance because it results in the best execution plan for consecutive execution. The highest performance increase for each data size is observed for the *mtmd-ml* policy at 1.83*x* (Figure 6.9(a) - Group-3), 1.27*x* (Figure 6.9(b) - Group-2), and 1.37*x* (Figure 6.9(c) - Group-3) for small, medium, and large sizes, respectively.

As shown in Figure 6.9, the *mtmd-ml* policy exhibits the highest performance across all data sizes and all groups of applications. The reason is that this policy leverages the ML-trained model to capture a large space of factors that can influence performance. In addition, there are cases where the consecutive execution on a single device (*DynRec* - baseline) results in higher performance than the concurrent execution on multiple devices with *fcfs*, *gpuprio*, or *cpuex*. For instance, Figure 6.9 shows that the applications in Group-1 can run significantly faster when they are executed consecutively on the Nvidia GPU rather than being concurrently executed across all available devices. The reason is that each application in Group-1 (i.e., DFT, BlackScholes, and Matrix Multiplication) is compute intensive and performs an order of magnitude faster on the Nvidia GPU than the other devices. Thus, the *fcfs*, *gpuprio*, or *cpuex* concurrent scheduling policies fail to outperform the baseline for these cases. On the contrary,



(a) Small Sizes.

(b) Medium Sizes.

(c) Large Sizes.

Figure 6.9: Achieved speedups for each group of applications and size configurations against the baseline Dynamic Reconfiguration (*DynRec*) for consecutive execution. Each bar presents the following policies: ML-based MTMD (*mtmd-ml*), First-Come-First-Served (*fcfs*), GPU Priority (*gpuprio*), and CPU Exclusion (*cpuex*).

mtmd-ml can achieve the baseline performance, as it accounts for the single context scenario during the ML model training. The only case that the *mtmd-ml* policy performs lower than the baseline is the medium size for Group-3 (Figure 6.9(b)). In this case, the trained ML model mispredicts and schedules the execution of the most of the compute-intensive task (i.e., NBody) in the small GPU (Intel UHD Graphics 630). Section 6.4.3 discusses the performance and precision analysis of the trained model in more detail.

Additionally, the remaining policies (*gpuprio*, *fcfs* and *cpuex*) show a diverse performance behaviour for the three groups of applications when running on the same data sizes. This indicates that the diversity across the applications that belong in the same group is high, and therefore, some of them can perform better on a GPU, while others can perform better on a CPU. For instance, Group-1 shows that the baseline outperforms all the remaining policies (i.e., *gpuprio*, *fcfs* and *cpuex*). The cause of this performance behaviour is the characteristics of the applications grouped together. These applications

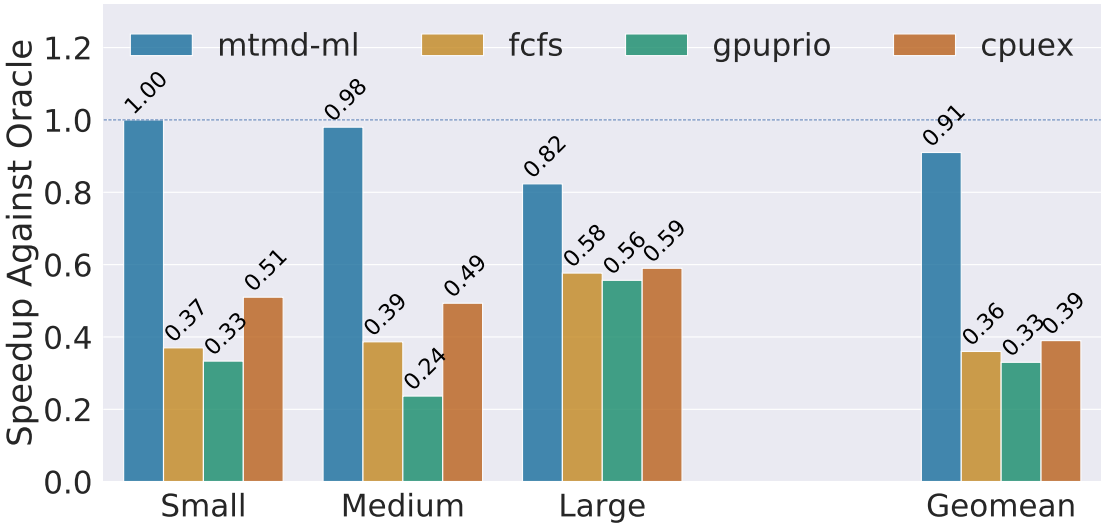


Figure 6.10: Comparison of the MTMD scheduling policies against the Oracle (peak performance).

are all compute-intensive and achieve high speedups when they are executed on the discrete GPU.

Group-2 exhibits higher performance than the baseline when the applications in this group are executed exclusively on the same GPUs (*cpuex* - orange bars), reaching up to 1.13*x* for medium size (Figure 6.9(b)). On the other hand, the performance of the *gpuprio*, *fcfs* and *cpuex* policies when running Group-3 is at the same range. A 0.08*x* performance difference is observed between *gpuprio* and *cpuex* for small sizes (Figure 6.9(a)), while a 0.17*x* difference is displayed between *fcfs* and *gpuprio*/*cpuex* for large sizes (Figure 6.9(c)). However, for medium sizes, *fcfs* achieves the highest performance among the MTMD policies, indicating that the GPUs are not the most suitable devices to execute for this range.

Finally, it is shown that the MTMD concurrent execution in conjunction with the ML-based scheduling policy (*mtmd-ml*) can increase the performance by up to 83% compared to the consecutive execution (*DynRec*).

6.4.2.2 Relative Performance versus Best Concurrent

To assess the performance of the MTMD scheduling policies against the maximum performance that can be achieved, the underlying experiments are augmented with an Oracle implementation. Therefore, the *mtmd-ml*, *fcfs*, *gpuprio* and *cpuex* policies are evaluated against the *Oracle* policy. *Oracle* represents the peak performance that can be achieved, as it is derived from the exhaustive exploration of all possible concurrent

execution plans of each group of benchmarks on the available hardware devices. Note that the diversity across the applications, along with the various data sizes, increases the exploration space significantly, and therefore, the decision of the *Oracle* policy may not be pragmatic for real applications. In fact, the execution of the applications in Group-2 for the large sizes takes 4.5 hours. Nonetheless, *Oracle* is the highest performing baseline to compare the performance of the MTMD policies in terms of concurrent execution.

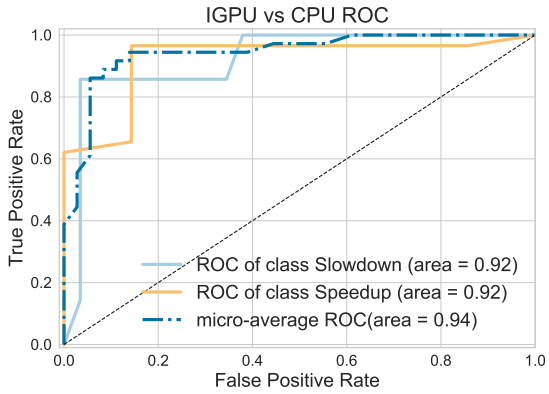
The left side of Figure 6.10 presents the comparative evaluation of the MTMD policies against *Oracle* for small, medium, and large data sizes, while the right side depicts their geometric mean. As Figure 6.10 shows, *mtmd-ml* is the best performing policy reaching up to 91% of the *Oracle*'s performance in average, followed by *cpuex* (39%) and *fcfs* (36%). The lowest average performance is observed for the *gpuprio* policy, due to the low performance of GPUs when running for small and medium data sizes.

6.4.3 Analysis of the ML Model used MTMD Scheduling

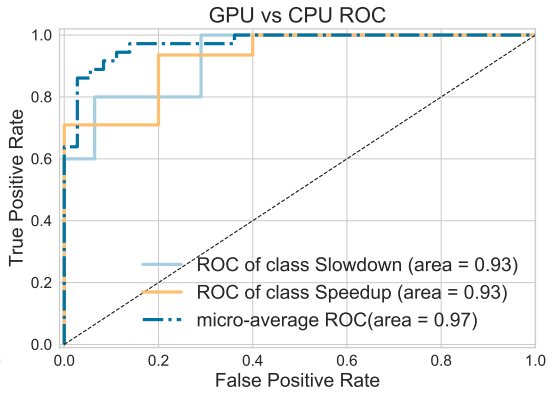
This section presents an analysis of the performance and successful task-device allocation of the trained MTMD machine learning model. In this work the metrics for performance evaluation are the *area under the ROC curve (AUC)* and the *F1-score*. The *AUC* is calculated as the integral of the ROC concerning the false positive rate over [0, 1], where a high *AUC* indicates the better prediction of the model.

Figure 6.11 presents the obtained *AUC* for the three classifiers that are employed by the model, as introduced in Section 6.3.4. In particular, the micro-average ROC that classifies the execution between two different types of devices is 0.94 (Figure 6.11(a)), 0.97 (Figure 6.11(b)) and 0.82 (Figure 6.11(c)) for the first, second and third classifier, respectively. Note[1], the micro-average ROC sums the true positives and false positives over the total target classes. Based on this metric, the second classifier (GPU-CPU) has the best performance, followed by the first (IGPU-CPU) and the third (GPU-IGPU) classifiers. This behaviour is also verified by investigating the confusion matrices in Tables 6.6, 6.7 and 6.8, which show that the third classifier mispredicted the IGPU over the GPU four out of 31 times. This is the cause of the misprediction that resulted in the low performance of Group-3 when *mtmd-ml* was used (Figure 6.9(b)), as the model decided to use the Intel Integrated GPU instead of the Nvidia GPU.

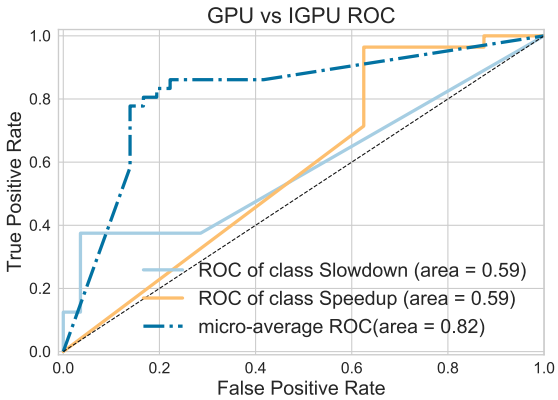
[1] `https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html`



(a) Classifier One



(b) Classifier Two



(c) Classifier Three

Figure 6.11: Offline training process and Online device allocation based on pre-trained model.

	Actual IGPU (1)	Actual CPU (0)
Predicted IGPU (1)	28	1
Predicted CPU (0)	1	6

Table 6.6: Confusion Matrix for Classifier One.

	Actual GPU (1)	Actual CPU (0)
Predicted GPU (1)	31	0
Predicted CPU (0)	2	3

Table 6.7: Confusion Matrix for Classifier Two.

	Actual GPU (1)	Actual IGPU (0)
Predicted GPU (1)	27	1
Predicted IGPU (0)	4	4

Table 6.8: Confusion Matrix for Classifier Three.

However, the overall decision is not severely influenced as the outcome on which device to execute takes into account all combinations of the classifiers. Finally, based on

the confusion matrices (Tables 6.6, 6.7 and 6.8), the *F1-score* (i.e., the harmonic mean of precision and recall) was computed for each classifier using the following formula:

In Equation 6.4, TP corresponds to true negative, FP to false positive and FN to false negative outcomes. Therefore, the final *F1-scores* are 0.95, 0.96 and 0.91 for the first, second and third classifier, respectively.

$$g(x) = \frac{TP}{TP + \frac{1}{2}(FP + FN)} \tag{6.4}$$

Moreover, another aspect of this approach is the inference time for the model. Through the results obtained during evaluation, on average, the inference time is 60ms.

6.5 Summary

In this chapter, a novel runtime augmented with an ML model that enables a heterogeneous managed runtime to support multiple-tasks on multiple-devices (MTMD) execution was presented. Compared to the prior work discussed in Section 3.4, this platform differs as to this date is the only system that targets Java applications for multiple-device heterogeneous execution. The complete mechanism uses parallel execution of bytecode interpreters to manage and execute arbitrary tasks across multiple OpenCL-compatible devices dynamically and concurrently. Moreover, to achieve an efficient device-task allocation, a machine learning approach with a multiple-classification architecture of Extra-Trees-Classifiers was employed. To provide the ML architecture with meaningful information, a feature extraction phase was designed in a way to capture them directly from the initial compiler graph.

The capabilities and adaptability of the MTMD mechanism with an intelligent scheduling process have undergone rigorous testing and evaluation. The complete toolchain was evaluated with applications split into three separate groups. Experimental results showcase performance improvements up 83% compared to all tasks running on the single highest performing device, while reaching up to 91% of the oracle performance.

The next chapter summarises the findings and contributions of this work. Also, it presents several future research directions that can extend the current infrastructure and developed concepts.

Chapter 7

Conclusions and Future Research Directions

High demand for increased computational capabilities and power efficiency has resulted in combining a plethora of heterogeneous hardware in modern computing systems. This trend has been followed by the emergence of heterogeneous programming frameworks that make diverse hardware more accessible to developers. To mitigate the steep learning curve of transitioning to heterogeneous resources, developers can nowdays utilise heterogeneous managed runtimes which abstract away the majority of required hardware knowledge. In this direction, this thesis utilises TornadoVM to demonstrate how a series of compiler and runtime optimisation can bridge the performance gap between conventional and heterogeneous virtual machines. As presented in the previous chapters, heterogeneous hardware can be accessible and achieve high performance without exposing the user to architecture-specific particularities.

This closing chapter summarises the contributions of this work in Section 7.1. Finally, Section 7.2 presents several future research directions that can capitalise on the findings of this thesis.

7.1 Summary

The contributions of this thesis can be summarised as follows:

Chapter 2 outlined the key concepts required to comprehend the contents of this thesis. In more detail, it presented information in the context of heterogeneous platforms, managed runtime systems, heterogeneous managed runtimes, and machine learning modelling. Chapter 3 presented and classified the state-of-art research work that is

aligned with the contributions of this thesis. Briefly, it summarised the work in three distinctive research areas: i) the FPGA execution in managed languages, (ii) optimising compilers for GPU code generation with memory hierarchy awareness, and (iii) multi-task scheduling on heterogeneous hardware.

Chapter 4 presented a practical approach that augments managed languages with seamless and efficient FPGA code execution. Moreover, it provided several specific specialisations, and optimisation phases that are transparently added to an existing open-source toolchain to increase the performance of unoptimised FPGA code. To achieve this result, it studied and outlined the engineering challenges and trade-offs when integrating the different toolchains with managed languages. The experimental platform used was the TornadoVM framework augmented with a two-stage compilation process and a set of FPGA-specific specialisation techniques. The proposed toolchain and optimisations were evaluated against a set of Java benchmarks executed on an Intel FPGA showcasing speedups up to 19.8×, 224×, and 3.82× over multi-threaded, sequential, and GPU-accelerated Java code, respectively. Also, this work provided the foundation that allowed the toolchain to host multiple state-of-the-art HLS compilers, such as Vivado [Cha16] HLS from Xilinx, and therefore target cloud-native solutions, such as the AWS (Amazon Web Services) cloud.

Chapter 5 presented an approach to efficiently exploit the memory hierarchy of GPUs from dynamically compiled languages. This is achieved by extending the capabilities of compiler snippets to express optimisations that improve data locality (e.g., local memory, loop tiling) by introducing the compositional compiler intrinsics. These can be parameterised and reused for different JIT compiler optimisations while having run-time information for the target GPU architecture. This investigative work outlined how a trade-off between compilation times and achieved performance can make it suitable for JIT-compiled languages. The evaluation was held against three GPU architectures, and the results indicate that it can achieve performance speedups of up to 1.58× and 2.5× for reduce and matrix operations, respectively. Moreover, when compared with manually written OpenCL code, the performance of the compiler extensions can achieve up to 97% of its performance while using the same set of optimisations. Most importantly, the performance stated above increases at no programmability costs, since they are transparently applied to unmodified user programs at compile-time without exposing parallel programming low-level notions to the user. Although the proposed technique has been researched in the context of TornadoVM, and inherently, the Graal compiler, it can be applied to other JIT compilers that target heterogeneous architectures.

Chapter 6 presented how a Multiple-Tasks on Multiple-Devices (MTMD) mechanism can provide seamless concurrent heterogeneous execution of Java programs. This mechanism was built by extending the virtualisation layer of TornadoVM with several novel components. In addition to a task dependency extraction, the novel components include a scalable and modular system that employs custom parallel bytecode interpreters that can scale for heterogeneous device orchestration. Although the extensions provided a system capable to deploy multiple tasks on multiple devices concurrently, performance was not guaranteed as it relies on efficient task-to-device allocation. Therefore, to ensure near-optimal device allocation, a custom ML-architecture of multiple classifiers was employed. The ML-architecture relies on features directly extracted from the compiler graph of a given input task. The capabilities of the ML-based allocation were showcased against a suite of 12 applications split into three separate groups and scheduled with various concurrent scheduling policies. Finally, the experimental evaluation showed performance improvements of up to 83% compared to the best single device, while reaching up to 91% of the oracle (i.e., the best task-to-device allocation) performance.

7.2 Future Research Directions

The work presented in this thesis can be used as a foundation to investigate several research directions, such as the MTMD execution with multi-backend awareness for languages running through Truffle [GSS+15, WWW+13]. A key aspect to assist further investigation is that the main experimental platform, TornadoVM, is an open-source and an actively maintained heterogeneous programming framework. The following points summarise a number of research directions:

- **Improved FPGA Utilisation:** As highlighted in Chapter 4 for the given set of benchmarks, up to one-third of the given FPGA area is utilised. Experimental investigation proved many directions towards mitigating this issue. Firstly, porting to Java an explicitly FPGA-oriented benchmark suite, such as Spector [GAMK16a] will provide further insights on the possible uses. Besides, the resulted resource utilisation can be improved by enabling FPGA-specific OpenCL features in the compiler, such as channels and pipes offered by Intel HLS [WOL+17]. Channels and pipes can allow the JIT compiler used in this work to combine kernels that sharing dependencies into a single source.

- **Adaptable Memory Tuning:** Chapter 5 provides insights on how one can exploit

the memory hierarchy of GPUs through JIT compilation techniques. However, one existing direction to this work is to adapt on-demand based on the input application needs instead of relying on the schematics of the input application. Therefore, during JIT compilation, decision making can allow the compiler IR graph to be adapted from allocation, thereby targeting the private memory to local or global based on specific sizes available during run-time.

- **Expressing Parallel Patterns with Compositional Intrinsics:** The technique described in detail through Chapter 5 can be leveraged to express various data parallel patterns, while targeting low-level memory optimisations. Candidate patterns can be various versatile groups that can merit from various levels of parallelism. For instance, stencil computations [RYQ11] for which fine-grained memory allocation due to irregular access patterns is a complex process. Besides, other data-parallel patterns, such as scan operations [SHGO11], scatter/gather [HGLS07] operations and parallel sorting algorithms [SJC17].

- **MTMD with Intelligent Multi-Backend Awareness:** The MTMD intelligent scheduling presented in Chapter 6 can be further improved to embed more fine-grained decision making. The ML-architecture can be extended to make decisions among different compiler backends (e.g., PTX [Nvi17], SPIR-V [Gro], x86) to ensure optimal device and architecture allocation for each application. Therefore, the end goal will be for a system capable of seamlessly offload workloads concurrently on multiple devices, while leveraging the optimal programming construct for each architecture. As the device prediction mechanism has its basis on features directly extracted from the Graal IR and runtime information, this process will enable multi-backend scheduling. The dispatching mechanism that isolates the execution in a device level-granularity can provide fine-control for multi-backend scheduling and performance.

- **MTMD with Intelligent Polyglot Awareness:** The aforementioned direction can provide the basis for multi-backend awareness; a complementary direction will provide polyglot support for MTMD. As the underlying experimental platform is based on the GraalVM, this work can be extended to make use of Truffle [GSS+15, WWW+13], an open-source library for building tools and programming languages implementations. Therefore, the goal will be to have a polyglot system with applications written in Java, Rudy, Python, and R will be scheduled for heterogeneous execution through their corresponding IRs.

Bibliography

[AAA16] Alejandro Acosta, Sergio Afonso, and Francisco Almeida. Extending paralldroid for the automatic generation of opencl code. In *Proceedings of the 4th International Workshop on OpenCL*, IWOCL '16, New York, NY, USA, 2016.

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2015.

[AAC+99] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing jalapeño in java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[AAZ+18] Qurrat Ul Ain, Saqib Ahmed, Abdullah Zafar, Muhammad Amir Mehmood, and Abdul Waheed. Analysis of Hotspot Methods in JVM for Best-Effort Run-Time Parallelization. In *Proceedings of the 9th International Conference on E-Education, E-Business, E-Management and E-Learning (IC4E)*, 2018.

[ABB+12] Joshua Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil Shukla. A compiler and runtime for heterogeneous computing. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 271–276, New York, NY, USA, 2012. Association for Computing Machinery.

[ABCR10] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2010.

[AGH05] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[Alt] Altimesh. Hybridizer essentials. `http://www.altimesh.com`.

[AMA+19] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4), July 2019.

[AMD16] AMD. Aparapi. `https://github.com/aparapi/aparapi`, 2016.

[APBcF16] Ashwin M. Aji, Antonio J. Peña, Pavan Balaji, and Wu chun Feng. Multicl: Enabling automatic scheduling for task-parallel workloads in opencl. *Parallel Computing*, 58:37–55, 2016.

[App] Apple. Metal. `https://developer.apple.com/metal/`.

[ARM21] ARM. Arm big.little. `https://www.arm.com/why-arm/technologies/big-little`, 2021.

[Ash08] Peter J. Ashenden. *"The Designer's Guide to VHDL, Volume 3, Third Edition (Systems on Silicon)"*. Morgan Kaufmann Publishers Inc., 2008.

[AUvAS+12] Shams A. H. Al Umairy, Alexander S. van Amesfoort, Irwan D. Setija, Martijn C. van Beurden, and Henk J. Sips. On the use of small 2d convolutions on gpus. In *Computer Architecture*, 2012.

[BA10] Kenneth P. Burnham and David Ray Anderson. *Model selection and multimodel inference: a practical information-theoretic approach*. Springer, New York, NY, 2. ed edition, 2010. OCLC: 846443242.

[BAM13] Rasmus Barringer and Tomas Akenine-Möller. A4: Asynchronous adaptive anti-aliasing using shared memory. *ACM Trans. Graph.*, 32(4), July 2013.

[BBK+08] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for Gpgpus. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS)*, 2008.

[BEP13] Kendrick Boyd, Kevin H. Eng, and C. David Page. Area under the precision-recall curve: Point estimates and confidence intervals. In *Proceedings of the 13th European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part III (ECMLPKDD)*, Berlin, Heidelberg, 2013.

[BFA14] Ioana Baldini, Stephen J. Fink, and Erik Altman. Predicting gpu performance from cpu runs using machine learning. In *Proceedings of the 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '14, page 254–261, USA, 2014. IEEE Computer Society.

[BGD94] Frederick J. Bremner, Stephen J. Gotts, and Dina L. Denham. Hinton diagrams: Viewing connection strengths in neural networks. *Behavior Research Methods, Instruments, Computers*, 26(2):215–218, 1994.

[BH98] P. Bellows and B. Hutchings. JHDL-an HDL for reconfigurable systems. In *IEEE 6th Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 1998.

[BH12] Nathan Bell and Jared Hoberock. Thrust: Productivity-Oriented Library for CUDA. *Astrophysics Source Code Library*, 2012.

[Bla03] Bruno Blanchet. Escape analysis for javatm: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, November 2003.

[BNS+21] Lorenz Braun, Sotirios Nikas, Chen Song, Vincent Heuveline, and Holger Fröning. A simple model for portable and fast prediction of execution time and power consumption of gpu kernels. *ACM Trans. Archit. Code Optim.*, 18(1), December 2021.

[Boh07] M. Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[BPCB10] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The Polyhedral Model Is More Widely Applicable Than You Think. In *International Conference on Compiler Construction (CC)*, 2010.

[Bra97] Andrew P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recogn.*, 30(7):1145–1159, July 1997.

[BRR+19] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019.

[BRS07] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. PLuTo: A Practical and Fully Automatic Polyhedral Parallelizer and Locality Optimizer. Technical Report OSU-CISRC-10/07-TR70, The Ohio State University, October 2007.

[BTL10] B. Betkaoui, D. B. Thomas, and W. Luk. Comparing performance and energy efficiency of fpgas and gpus for high productivity computing. In *2010 International Conference on Field-Programmable Technology*, pages 94–101, 2010.

[BVR+12] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel. In *Proceedings of the 49th Annual Design Automation Conference DAC*. ACM Press, 2012.

[C+15] Francois Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[CA12] Linchuan Chen and Gagan Agrawal. Optimizing mapreduce for gpus with effective shared memory usage. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2012.

[CBHK02] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321–357, June 2002.

[CBM+09] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[CC19] Anthony M. Cabrera and Roger D. Chamberlain. Exploring portability and performance of opencl fpga kernels on intel harpv2. In *Proceedings of the International Workshop on OpenCL*, IWOCL'19, New York, NY, USA, 2019. Association for Computing Machinery.

[CDK+01] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[CDL13] Eric S. Chung, John D. Davis, and Jaewon Lee. LINQits: Big Data on Little Clients. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[CFP+18a] James Clarkson, Juan Fumero, Michail Papadimitriou, Maria Xekalaki, and Christos Kotselidis. Towards practical heterogeneous virtual machines. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, Programming'18 Companion, page 46–48, New York, NY, USA, 2018. Association for Computing Machinery.

[CFP+18b] James Clarkson, Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, Christos Kotselidis, and Mikel Lujan. Exploiting

high-performance heterogeneous hardware for java programs using graal. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang '18, New York, NY, USA, 2018. Association for Computing Machinery.

[CFP+18c] James Clarkson, Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, Christos Kotselidis, and Mikel Luján. Exploiting High-Performance Heterogeneous Hardware for Java Programs Using Graal. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang)*, 2018.

[CGK+13] Albert Cohen, Tobias Grosser, Paul Kelly, J. Ramanujam, Ponnuswamy Sadayappan, and Sven Verdoolaege. Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles to Reconcile Parallelism and Locality, avoiding Divergence and Load Imbalance. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing (GPGPU)*, 2013.

[Cha16] Sudipto Chakraborty. Vivado design tools. In *Designing with Xilinx® FPGAs*, pages 17–21. Springer International Publishing, October 2016.

[CHL+17] P. Colangelo, R. Huang, E. Luebbers, M. Margala, and K. Nealis. Fine-Grained Acceleration of Binary Neural Networks Using Intel® Xeon® Processor with Integrated FPGA. In *IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017.

[CKBL15] James Clarkson, Christos Kotselidis, Gavin Brown, and Mikel Luján. Boosting java performance using gpgpus, 2015.

[Cla19] James Clarkson. *Compiler and Runtime Support for Heterogeneous Programming*. PhD thesis, The University of Manchester, 08 2019.

[CLL+15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, 2015.

[CMJ+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[Col91] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

[CP95] Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, page 35–49, New York, NY, USA, 1995. Association for Computing Machinery.

[CPB+18] P. Caldeira, J. C. Penha, L. Braganca, R. Ferreira, J. A. M. Nacif, R. Ferreira, and F. M. Q. Pereira. From Java to FPGA: an Experience with the Intel HARP System. In *30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2018.

[CPSL15] Chris Cummins, Pavlos Petoumenos, Michel Steuwer, and Hugh Leather. Autotuning opencl workgroup size for stencil patterns. *CoRR*, abs/1511.02490, 2015.

[CPWL17] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232, 2017.

[CTD+17] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood. A Pythonic Approach for Rapid Hardware Prototyping and Instrumentation. In *27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[DBAS18] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio. A unified backend for targeting fpgas from dsls. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2018.

[DCR+12] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen 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 (PLDI)*, PLDI '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[DGHGL+19] Simon Garcia De Gonzalo, Sitao Huang, Juan Gómez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu. Automatic generation of warp-level primitives and atomic instructions for fast and portable parallel reduction on gpus. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CG0)*, 2019.

[DGY+74] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[DSW+13] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck. Graal IR: An Extensible Declarative Intermediate Representation. In *Asia-Pacific Programming Languages and Compilers (APPLC)*, 2013.

[DWS+13] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, page 1–10, New York, NY, USA, 2013. Association for Computing Machinery.

[DYS+12] P. Di, D. Ye, Y. Su, Y. Sui, and J. Xue. Automatic Parallelization of Tiled Loop Nests with Enhanced Fine-Grained Parallelism on GPUs. In *41st International Conference on Parallel Processing (ICPP)*, 2012.

[Ecl] Eclipse. Eclipse openj9. https://www.eclipse.org/openj9/docs/.

[Far10] Frank A. Farris. The gini index and measures of inequality. *The American Mathematical Monthly*, 117(10):851–864, 2010.

[FBC+09] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying magic: High-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, page 81–90, New York, NY, USA, 2009. Association for Computing Machinery.

[FK18] Juan Fumero and Christos Kotselidis. Using Compiler Snippets to Exploit Parallelism on Heterogeneous Hardware: A Java Reduction Case Study. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL)*, 2018.

[Fla06] David Flanagan. *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2006.

[FPZ+19] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. Dynamic application reconfiguration on heterogeneous hardware. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, page 165–178, New York, NY, USA, 2019. Association for Computing Machinery.

[FRSD15] Juan José Fumero, Toomas Remmelg, Michel Steuwer, and Christophe 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 (PPPJ)*, PPPJ '15. Association for Computing Machinery, 2015.

[FSSD17] Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe 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*, VEE '17, 2017.

[FSV14] Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. Aristotle: A performance impact indicator for the opencl kernels using local memory. *Sci. Program.*, 22(3):239–257, July 2014.

[Fum17] Juan Fumero. *Accelerating interpreted programming languages on GPUs with just-in-time compilation and runtime optimisations*. PhD thesis, The University of Edinburgh, 11 2017.

[GAMK16a] Q. Gautier, A. Althoff, Pingfan Meng, and R. Kastner. Spector: An OpenCL FPGA benchmark suite. In *International Conference on Field-Programmable Technology (FPT)*, 2016.

[GAMK16b] Quentin Gautier, Alric Althoff, Pingfan Meng, and Ryan Kastner. Spector: An opencl fpga benchmark suite. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 141–148, 2016.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, October 2007.

[GCH+14] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, page 66–75, New York, NY, USA, 2014. Association for Computing Machinery.

[GCK+13] Tobias Grosser, Albert Cohen, Paul H. J. Kelly, J. Ramanujam, P. Sadayappan, and Sven Verdoolaege. Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU)*, 2013.

[GEW06] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Mach. Learn.*, 63(1):3–42, April 2006.

[GGKSC13] Scott Grauer-Gray, William Killian, Robert Searles, and John Cavazos. Accelerating financial applications on the gpu. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, page 127–136, New York, NY, USA, 2013. Association for Computing Machinery.

[GGXS+12] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10, 2012.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1996.

[GLC+11] V. M. Garcia, A. Liberos, A. M. Climent, A. Vidal, J. Millet, and A. González. An adaptive step size gpu ode solver for simulating the electric cardiac activity. In *2011 Computing in Cardiology*, pages 233–236, 2011.

[GLD+08] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete fourier transforms on graphics processors. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008.

[GLS99] Steve Guccione, Delon Levi, and Prasanna Sundararajan. JBits: Java based interface for reconfigurable computing. In *Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999.

[GO11] Dominik Grewe and Michael F. P. O'Boyle. A static task partitioning approach for heterogeneous systems using OpenCL. In *Lecture Notes in Computer Science*, pages 286–305. Springer Berlin Heidelberg, 2011.

[Gooa] Google. Turbofan. `https://v8.dev/docs/turbofan`.

[Goob] Google. What is v8? `https://v8.dev/`.

[Gro] Khronos Group. Khronos spir-v. https://www.khronos.org/registry/spir-v/.

[GSK+20] Anirban Ghose, Siddharth Singh, Vivek Kulaharia, Lokesh Dokara, Srijeeta Maity, and Soumyajit Dey. Pyschedcl: Leveraging concurrency in heterogeneous data-parallel systems, 2020.

[GSS+15] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. High-performance cross-language interoperability in a multi-language runtime. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, page 78–90, New York, NY, USA, 2015. Association for Computing Machinery.

[GWO13] D. Grewe, Z. Wang, and M. F. P. O'Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10, 2013.

[HBD+13] Sylvain Henry, Denis Barthou, Alexandre Denis, Raymond Namyst, and Marie-Christine Counilh. SOCL: An OpenCL Implementation with Automatic Multi-Device Adaptation Support. Research Report RR-8346, INRIA, August 2013.

[HGLS07] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2007.

[HHA+18] Mohamed W. Hassan, Ahmed E. Helal, Peter M. Athanas, Wu-Chun Feng, and Yasser Y. Hanafy. Exploring fpga-specific optimizations for irregular opencl applications. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2018.

[HHWG12] Huynh Phung Huynh, Andrei Hagiescu, Weng-Fai Wong, and Rick Siow Mong Goh. Scalable framework for mapping streaming applications onto multi-gpu systems. In *Proceedings of the 17th ACM*

SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12, page 1–10, New York, NY, USA, 2012. Association for Computing Machinery.

[HLK+20] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Sergei Gorlatch, and Michel Steuwer. A Language for Describing Optimization Strategies, 2020.

[HM13] Haibo He and Yunqian Ma, editors. *Imbalanced learning: foundations, algorithms, and applications*. John Wiley & Sons, Inc, Hoboken, New Jersey, 2013.

[Ho95] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282 vol.1, 1995.

[HR15] A. Huseinović and S. Ribić. Benchmark comparison of computing the mandelbrot set in opencl. In *2015 23rd Telecommunications Forum Telfor (TELFOR)*, pages 994–997, 2015.

[HS99] Geoffrey E Hinton and Terrence J Sejnowski. *Unsupervised learning foundations of neural computation*. 1999. OCLC: 1227497992.

[HSS+18] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. High Performance Stencil Code Generation with Lift. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2018.

[IHKS15] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar. Compiling and Optimizing Java 8 Programs for GPU Execution. In *International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[IKL+17] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017.

[Inta] Intel. GPGPU: Intel Iris Xe MAX graphics. `https://dgpu-docs.intel.com/devices/iris-xe-max-graphics/index.html`.

[Intb] Intel. List of HD Graphics for Intel Core i9, i7, i5, and i3. https://www.intel.com/content/www/us/en/support/articles/000057924/processors/intel-core-processors.html.

[Intc] Intel. oneAPI Specification. https://spec.oneapi.com/versions/latest/index.html.

[Intd] Intel. The Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[Int19] Intel FPGA SDK for OpenCL. Pro Edition. Version 19.1 Release Notes, 2019. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/rn/archives/rn_aocl-19-1.pdf.

[Int20] Intel. Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA. https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/acceleration-card-arria-10-gx/documentation.html, Feb 2020.

[Int21] Intel. Introduction to the acceleration stack for Intel Xeon CPU with FPGAs. https://www.intel.com/content/www/us/en/programmable/support/training/course/oaccelintro.html, Apr 2021.

[Ion17] V. M. Ionescu. Cpu and gpu gray scale image conversion on mobile platforms. In *2017 9th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pages 1–6, 2017.

[JCBM16] Ivan Jibaja, Ting Cao, Stephen M. Blackburn, and Kathryn S. McKinley. Portable performance on asymmetric multicore processors. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 24–35, New York, NY, USA, 2016. Association for Computing Machinery.

[Jol86] I. T. Jolliffe. *Principal Component Analysis and Factor Analysis*, pages 115–128. Springer New York, New York, NY, 1986.

[JPT+13] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. Inspire: The insieme parallel intermediate representation. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, page 7–18. IEEE Press, 2013.

[JS19] Dejice Jacob and Jeremy Singer. Alpyna: Acceleration of loops in python for novel architectures. In *6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY)*, pages 25–34. ACM Press, June 2019.

[JTS19a] Dejice Jacob, Phil Trinder, and Jeremy Singer. Python programmers have GPUs too: Automatic Python loop parallelization with staged dependence analysis. In *Proceedings of the Dynamic Languages Symposium*, 2019.

[JTS19b] Dejice Jacob, Phil Trinder, and Jeremy Singer. Python programmers have gpus too: Automatic python loop parallelization with staged dependence analysis. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2019, page 42–54, New York, NY, USA, 2019. Association for Computing Machinery.

[KAA+19] Yasir Noman Khalid, Muhammad Aleem, Usman Ahmed, Muhammad Arshad Islam, and Muhammad Azhar Iqbal. Troodon: A machine-learning based load-balancing application scheduler for cpu–gpu system. *Journal of Parallel and Distributed Computing*, 132:79 – 94, 2019.

[KB16] N. Kapre and S. Bayliss. Survey of Domain-Specific Languages for FPGA Computing. In *26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[KCR+17a] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. Heterogeneous managed runtime systems: A computer vision case study. VEE '17, page 74–82, New York, NY, USA, 2017. Association for Computing Machinery.

[KCR+17b] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. Heterogeneous Managed Runtime

Systems: A Computer Vision Case Study. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2017.

[KDA+20] C. Kotselidis, S. Diamantopoulos, O. Akrivopoulos, V. Rosenfeld, K. Doka, H. Mohammed, G. Mylonas, V. Spitadakis, and W. Morgan. Efficient compilation and execution of jvm-based data processing frameworks on heterogeneous co-processors. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 175–179, 2020.

[KFAB16] Konstantinos Krommydas, Wu-Chun Feng, Christos D. Antonopoulos, and Nikolaos Bellas. Opendwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures. *J. Signal Process. Syst.*, 85(3):373–392, December 2016.

[KFP+18] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and et al. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[KGCF13] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, page 149–160, New York, NY, USA, 2013. Association for Computing Machinery.

[KKRS14] Athanasios Konstantinidis, Paul H. J. Kelly, J. Ramanujam, and P. Sadayappan. Parametric GPU Code Generation for Affine Loop Programs. In *Languages and Compilers for Parallel Computing (LCPC)*, 2014.

[Kot07] S B Kotsiantis. Supervised machine learning: A review of classification techniques. *Informatica*, 31(3):249–268, 2007.

[KPZ+16] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. Automatic Generation of Efficient Accelerators for

Reconfigurable Hardware. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[KS16] Asterios Katsifodimos and Sebastian Schelter. Apache flink: Stream analytics at scale. In *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*, pages 193–193, 2016.

[KSL+12] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. Snucl: An opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, page 341–352, New York, NY, USA, 2012. Association for Computing Machinery.

[KSRT+19] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. A Code Generator for High-Performance Tensor Contractions on GPUs. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019.

[KWM+08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspot™ client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1), May 2008.

[LA04] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2004.

[LG99] Allen Leung and Lal George. Static single assignment form for machine code. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, page 204–214, New York, NY, USA, 1999. Association for Computing Machinery.

[LHK09] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM Press, 2009.

[LHWW21] Zhifang Li, Mingcong Han, Shangwei Wu, and Chuliang Weng. Shadowvm: Accelerating data plane for data analytics with bare metal cpus and gpus. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 147–160, New York, NY, USA, 2021. Association for Computing Machinery.

[Lom11] Chris Lomont. Introduction to intel advanced vector extensions. intel white paper, 2011.

[Ltd] Arm Ltd. The arm mali-g78 gpu. `https://developer.arm.com/ip-products/graphics-and-multimedia/mali-gpus/mali-g78-gpu`.

[Mat08] Bernd Mathiske. The maxine virtual machine and inspector. In *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA Companion '08, page 739–740, New York, NY, USA, 2008. Association for Computing Machinery.

[Max11] Maxeler Technologies. MaxCompiler White Paper. `https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf`, 2011.

[MCC18] Thierry Moreau, Tianqi Chen, and Luis Ceze. Leveraging the VTA-TVM Hardware/Software Stack for FPGA Acceleration of 8-bit ResNet-18 Inference. In *Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-designing Pareto-efficient Deep Learning*, 2018.

[Mit97] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., USA, 1 edition, 1997.

[MO14] Christos Margiolas and Michael F. P. O'Boyle. Portable and transparent host-device communication optimization for gpgpu environments. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, page 55–65, New York, NY, USA, 2014. Association for Computing Machinery.

[MO16] Christos Margiolas and Michael F. P. O'Boyle. Portable and transparent software managed scheduling on accelerators for fair resource sharing. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 82–93, New York, NY, USA, 2016. Association for Computing Machinery.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114, April 1965.

[Moza] Mozilla. Ionmonkey. `https://wiki.mozilla.org/IonMonkey`.

[Mozb] Mozilla. Spidermonkey. `https://firefox-source-docs.mozilla.org/js/index.html`.

[MSG+18] S. Margerm, A. Sharifian, A. Guha, A. Shriraman, and G. Pokam. TAPAS: Generating Parallel Accelerators from Parallel Programs. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[NBB19] R. Nozal, J. L. Bosque, and R. Beivide. Towards co-execution on commodity heterogeneous systems: Optimizations for time-constrained scenarios. *2019 International Conference on High Performance Computing and Simulation (HPCS)*, pages 628–635, 2019.

[NBZ+15] Luigi Nardi, Bruno Bodin, M. Zeeshan Zia, John Mawer, Andy Nisbet, Paul H. J. Kelly, Andrew J. Davison, Mikel Luján, Michael F. P. O'Boyle, Graham Riley, Nigel Topham, and Steve Furber. Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, May 2015. arXiv:1410.2167.

[NSP+16] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

[NVF20] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[Nvi17] Nvidia. Parallel thread execution isa. https://docs.nvidia.com/pdf/ptx_isa_5.0.pdf, 2017.

[OM99] D. Opitz and R. Maclin. Popular ensemble methods: An empirical study. *Journal of Artificial Intelligence Research*, 11:169–198, August 1999.

[OPWL15] William F. Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Fast automatic heuristic construction using active learning. In *Languages and Compilers for Parallel Computing (LCPC)*, pages 146–160. Springer International Publishing, 2015.

[Oraa] Oracle. The java virtual machine instruction set. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html.

[Orab] Oracle. Project sumatra. https://openjdk.java.net/projects/sumatra/.

[Ora14] Oracle. Loop optimizations in Hotspot Server VM Compiler (C2). https://wiki.openjdk.java.net/pages/viewpage.action?pageId=20415918, 2014.

[OYIY18] Satoshi Ohshima, Ichitaro Yamazaki, Akihiro Ida, and Rio Yokota. Optimization of hierarchical matrix computation on gpu. In *Supercomputing Frontiers*. Springer International Publishing, 2018.

[Pap16] Michail Papadimitriou. Accelerating computational finance simulations; with opencl. Master's thesis, Tu Delft, 2016.

[PCC+14] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA)*, 2014.

[PCV17] Michail Papadimitriou, Joris Cramwinckel, and Ana Lucia Varbanescu. Speed-up computational finance simulations with opencl on intel xeon phi. In *Euro-Par 2016: Parallel Processing Workshops*, pages 199–208, Cham, 2017. Springer International Publishing.

[PDAS20] Alberto Parravicini, Arnaud Delamare, Marco Arnaboldi, and Marco D. Santambrogio. Dag-based scheduling with resource sharing for multi-task applications in a polyglot gpu runtime, 2020.

[Per93] Douglas L. Perry. *VHDL*. McGraw-Hill, Inc., 2 edition, 1993.

[PFK18] Michail Papadimitriou, Juan Fumero, and Christos Kotselidis. Exploiting programmability of fpgas through managed runtime systems. *International Summer School on Advanced Computer Architecture and Compilation for High-performance Embedded Systems (ACACES)*, 2018.

[PFS+20] Michail Papadimitriou, Juan Fumero, Athanasios Stratikopoulos, Foivos S. Zakkak, and Christos Kotselidis. Transparent compiler and runtime specializations for accelerating managed languages on fpgas. *The Art, Science, and Engineering of Programming*, 5(2), Oct 2020.

[PFSK19] M. Papadimitriou, J. Fumero, A. Stratikopoulos, and C. Kotselidis. Towards prototyping and acceleration of java programs onto intel fpgas. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019.

[PFSK21] Michail Papadimitriou, Juan Fumero, Athanasios Stratikopoulos, and Christos Kotselidis. Automatically exploiting the memory hierarchy of gpus through just-in-time compilation. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2021, page 57–70, New York, NY, USA, 2021. Association for Computing Machinery.

[PG14] Prasanna Pandit and R. Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, page 273–283, New York, NY, USA, 2014. Association for Computing Machinery.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[PJH09] DP Playne, MGB Johnson, and KA Hawick. Benchmarking gpu devices with n-body simulations. In *Proc. 2009 International Conference on Computer Design (CDES 09) July, Las Vegas, USA., no. CSTN-077*, 2009.

[PMF+21] Michail Papadimitriou, Eleni Markou, Juan Fumero, Athanasios Stratikopoulos, Florin Blanaru, and Christos Kotselidis. Multiple-tasks on multiple-devices (mtmd): Exploiting concurrency in heterogeneous managed runtimes. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2021, page 125–138, New York, NY, USA, 2021. Association for Computing Machinery.

[PSFW12] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch. Rootbeer: Seamlessly Using GPUs from Java. In *IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, 2012.

[PVC01a] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspottm server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, page 1, USA, 2001. USENIX Association.

[PVC01b] Michael Paleczny, Christopher Vick, and Cliff Click. The Java hotspotTM Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, 2001.

[Qui86] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986.

[RBD+11] Vidya Rajagopalan, Vamsi Boppana, Sandeep Dutta, Brad Taylor, and Ralph Wittig. Xilinx zynq-7000 EPP: An extensible processing platform family. In *2011 IEEE Hot Chips 23 Symposium (HCS)*. IEEE, August 2011.

[Red16] Joseph Redmon. Darknet: Open source neural networks in c. `http://pjreddie.com/darknet/`, 2013–2016.

[RK16] Reuven Y. Rubinstein and Dirk P. Kroese. *Simulation and the Monte Carlo Method*. Wiley Publishing, 3rd edition, 2016.

[RKAS+17a] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61(1):106–115, December 2017.

[RKAS+17b] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61(1):106–115, December 2017.

[RKBA+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman 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 (PLDI)*, 2013.

[RKH+11] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. A Programming Language Interface to Describe Transformations and Code Generation. In *Languages and Compilers for Parallel Computing (LCPC)*, 2011.

[RLSD16] Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. Performance Portable GPU Code Generation for Matrix Multiplication. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit (GPGPU)*, 2016.

[RM20] Lukas Stadler Rene Mueller. Grcuda. https://github.com/NVIDIA/grcuda, 2020.

[Ros] John Rose. JEP 243: Java-Level JVM Compiler Interface. https://openjdk.java.net/jeps/243.

[RYC+13] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[RYQ11] Shah M. Faizur Rahman, Qing Yi, and Apan Qasem. Understanding stencil code performance on multicore architectures. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, New York, NY, USA, 2011. Association for Computing Machinery.

[SAGL18] Andreas Simbürger, Sven Apel, Armin Größlinger, and Christian Lengauer. PolyJIT: Polyhedral optimization just in time. *International Journal of Parallel Programming*, 47(5-6):874–906, August 2018.

[SBL+14] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 2014.

[SCN+15] O. Segal, P. Colangelo, N. Nasiri, Z. Qian, and M. Margala. AparapiUCores: A High Level Programming Framework for Unconventional Cores. In *IEEE High Performance Extreme Computing Conference (HPEC)*, 2015.

[SD09] Richard M. Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Scotts Valley, CA, 2009.

[SEEZ19] Kholoud Shata, Marwa K. Elteir, and Adel A. EL-Zoghabi. Optimized implementation of OpenCL kernels on FPGAs. *Journal of Systems Architecture*, 97:491–505, August 2019.

[SG08] Satnam Singh and David J. Greaves. Kiwi: Synthesis of FPGA circuits from parallel programs. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, April 2008.

[SGS10] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, May 2010.

[SHGO11] Shubhabrata Sengupta, Mark Harris, Michael Garland, and John Owens. *Efficient Parallel Scan Algorithms for GPUs*, pages 413–442. CRC Press, 01 2011.

[SJC17] Dhirendra Pratap Singh, Ishan Joshi, and Jaytrilok Choudhary. Survey of GPU based sorting algorithms. *International Journal of Parallel Programming*, 46(6):1017–1034, April 2017.

[SKG11] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011.

[SMCW] Oren Segal, Martin Margala, Sai Rahul Chalamalasetti, and Mitch Wright. High Level Programming for Heterogeneous Architectures. In *1st International Workshop on FPGAs for Software Programmers (FSP 2014)*.

[SMSF18] S. Skalicky, J. Monson, A. Schmidt, and M. French. Hot&Spicy: Improving Productivity with Python and HLS for FPGAs. In *IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018.

[SOV+20] Athanasios Stratikopoulos, Mihai-Cristian Olteanu, Ian Vaughan, Zoran Sevarac, Nikos Foutris, Juan Fumero, and Christos Kotselidis. Transparent acceleration of java-based deep learning engines. In *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*, MPLR 2020, page 73–79, New York, NY, USA, 2020. Association for Computing Machinery.

[SPB+17] Amit Kumar Singh, Alok Prakash, Karunakar Reddy Basireddy, Geoff V. Merrett, and Bashir M. Al-Hashimi. Energy-efficient run-time mapping and thread partitioning of concurrent opencl applications on cpu-gpu mpsocs. *ACM Trans. Embed. Comput. Syst.*, 16(5s), September 2017.

[SRD16] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Matrix multiplication beyond auto-tuning. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems - CASES '16*. ACM Press, 2016.

[SRD17] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, February 2017.

[STDP19] Alberto Scionti, Olivier Terzo, Karim Djemame, and Clara Pezuela. *Heterogeneous Computing Architecture - Challenges and Vision*. CRC Press, 09 2019.

[SWU+15] Doug Simon, Christian Wimmer, Bernhard Urban, Gilles Duboscq, Lukas Stadler, and Thomas Würthinger. Snippets: Taking the High Road to a Low Level. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2015.

[Tha19] Mohit Thakkar. *Beginning Machine Learning in IOS: CoreML Framework*. APress, 1st edition, 2019.

[TM96] Donald E. Thomas and Philip R. Moorby. *The Verilog® Hardware Description Language*. Springer US, 1996.

[TW17] T. N. Theis and H. . P. Wong. The end of moore's law: A new beginning for information technology. *Computing in Science Engineering*, 19(2):41–50, 2017.

[TWSC10] Ben L. Titzer, Thomas Wurthinger, Doug Simon, and Marcelo Cintra. Improving compiler-runtime separation with xir. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, page 39–50, New York, NY, USA, 2010. Association for Computing Machinery.

[UGT09] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software pipelined execution of stream programs on gpus. In *2009 International Symposium on Code Generation and Optimization*, CGO '09, pages 200–209, 2009.

[VCJC+13] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[VD08] V. Volkov and J.W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, November 2008.

[VRDJ95] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[WGO15] Zheng Wang, Dominik Grewe, and Michael F. P. O'boyle. Automatic and portable mapping of data parallel programs to OpenCL for GPU-based heterogeneous systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–26, January 2015.

[WHU17] Hasitha Muthumala Waidyasooriya, Masanori Hariyama, and Kunio Uchiyama. *Design of FPGA-Based Computing Systems with OpenCL*. Springer Publishing Company, Incorporated, 1 edition, 2017.

[WHVDV+13] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[WNDS99] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[WO18] Zheng Wang and Michael O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.

[WOL+17] Dennis Weller, Fabian Oboril, Dimitar Lukarski, Juergen Becker, and Mehdi Tahoori. Energy efficient scientific computing on fpgas using opencl. In *Proceedings of the 2017 International Symposium on Field-Programmable Gate Arrays (FPGA)*, New York, NY, USA, 2017. Association for Computing Machinery.

[WPHZ17] Zeke Wang, Johns Paul, Bingsheng He, and Wei Zhang. Multikernel data partitioning with channel on opencl-based fpgas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(6):1906–1918, 2017.

[WPSM10] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU Microarchitecture Through Microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, 2010.

[WWO14] Yuan Wen, Zheng Wang, and Michael F. P. O'Boyle. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *2014 21st International Conference on High Performance Computing (HiPC)*. IEEE, December 2014.

[WWS10] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, page 10–19, New York, NY, USA, 2010. Association for Computing Machinery.

[WWW+13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario 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*, Onward! '13, page 187–204, New York, NY, USA, 2013. ACM.

[XFK18] Maria Xekalaki, Juan Fumero, and Christos Kotselidis. Challenges and proposals for enabling dynamic heterogeneous execution of big data frameworks. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, December 2018.

[YWTC15] Yi-Ping You, Hen-Jung Wu, Yeh-Ning Tsai, and Yen-Ting Chao. Virtcl: A framework for opencl device abstraction and management. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, page 161–172, New York, NY, USA, 2015. Association for Computing Machinery.

[YXKZ10] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[ZBMS10] Dimitrios Ziakas, Allen Baum, Robert A. Maddox, and Robert J. Safranek. Intel quickpath interconnect architectural features supporting scalable system architectures. In *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects (HOTI)*, USA, 2010. IEEE Computer Society.

[ZLG12] Wojciech Zaremba, Yuan Lin, and Vinod 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*, 2012.

[Zoh18] Hamid Reza Zohouri. High performance computing with fpgas and opencl. *CoRR*, abs/1810.09773, 2018.

[ZSC13] Yao Zhang, Mark Sinclair, and Andrew A. Chien. Improving performance portability in OpenCL programs. In *Lecture Notes in Computer Science*, pages 136–150. Springer Berlin Heidelberg, 2013.

[ZXW+16] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.