UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Using Just-in-Time Code Generation to Transparently Accelerate Applications in Heterogeneous Systems

Dissertation

A thesis submitted to the
Faculty of Electrical Engineering, Computer Science and Mathematics
of
Paderborn University
in partial fulfilment of the requirements for the
degree of *Dr. rer. nat.*

by
Gavin Vaz

Paderborn, Germany
11th July 2019

Acknowledgements

My PhD journey has been an unforgettable and life-changing experience, which would not have been made possible without the help and support from so many people. First and foremost, I would like to thank Prof. Dr. Christian Plessl for being my academic adviser during my PhD studies. I cannot imagine having a better adviser and I am very grateful for his guidance, encouragement and support both professionally and personally!

Furthermore, I would like to thank:

- Prof. Dr. Marco Platzner, for his support during my PhD studies and for serving as a reviewer for my dissertation.
- Prof. Dr. Franz Rammig , Prof. Dr. Anthony Anjorin and Dr. Peter Pfahler for serving on my oral examination committee.
- All my colleagues from the High-Performance IT Systems Group, the Paderborn Center for Parallel Computing (PC^2) and the Computer Engineering Group with whom I have had the pleasure to work with. I will definitely miss the entertaining water cooler banter.
- My office mate Heinrich Riebler for our extensive collaborative research, our "Operation Tomato" office experiment and key insider tips into the German way of life.
- Dr. Tobias Kenter for his guidance and joint collaboration on several research interests.
- Achim Lösch for his unmatched technical expertise and indispensable advice.
- Michael Laß, Paolo Gorlani and Linus Witschen for their invaluable feedback.
- The European Union Seventh Framework Programme (FP7) and the Deutsche Forschungsgemeinschaft (DFG) for funding different phases of my research in the "SAVE" project and in the Collaborative Research Centre 901 "On-The-Fly Computing" project.
- All my colleagues in the "SAVE" project for their insightful discussions.

Finally, I would like to thank my family. This journey would not have been possible without their support. I am grateful to my parents Ida and Cecil, who have always been there for me and encouraged me in all my pursuits and inspired me to follow my dreams. A special thank you to Sunshine Bujok for all her love and support and for always believing in me. Last but not least, I would like to thank the Biedemann family for treating me as a part of the family and making me feel at home in Germany.

Abstract

In the high performance computing domain, data parallel workloads are driving architectures toward energy efficient heterogeneous systems. These systems are capable of executing computationally intensive tasks on accelerators designed to maximize data throughput. However, integrating different accelerators with diverse architectures and memory hierarchies into the same compute node increases the complexity of such systems, making it difficult to achieve an optimal performance/energy trade-off on heterogeneous systems. The SAVE Heterogeneous System Architecture (*saveHSA*) tries to solve this by using a smart self-adaptive layer to autonomously offload workloads to available resources by taking into account the varying workloads, application goals, system constraints and resource availability. However, to benefit from this, application developers need to port their applications to different architectures, which requires not only application and domain-specific knowledge, but also the understanding of different accelerator architectures, resulting in increased design effort and overall costs.

To help simplify the porting process and assist the application developer in generating parallelized code, current state-of-the-art parallelization techniques use pragma-based approaches, source-to-source transformations, template libraries, new programming languages and various semi and fully-automatic parallelization compilers and tools. However, most of these approaches are not completely automatic and require varying levels of effort from the application developer, while most of the fully-automated tools are unable to target multiple heterogeneous accelerators. Among the different parallel programming models, OpenCL is the most portable one. However, it requires a lot of manual effort by the application developer to parallelize the application by first generating parallel OpenCL kernel code and then the corresponding OpenCL host code, which is a time intensive and error-prone process.

In this thesis, we improve the state-of-the-art by developing a novel automatic and transparent parallelization approach known as the Runtime and Just-in-Time Compilation System (RTCS), which is capable of transparently porting sequential programs to different heterogeneous multi-accelerator architectures via OpenCL. Using the RTCS allows the *saveHSA* to efficiently manage heterogeneous systems by Just-in-Time (JIT) generating accelerator specific code. The RTCS parallelizes the application by automatically detecting and transforming suitable data-parallel loops into independent OpenCL kernels. The corresponding OpenCL host code required to setup the OpenCL device, create OpenCL buffers, transfer data to the device and launch the OpenCL kernel on the device is also automatically generated. Additionally, the RTCS applies data transfer optimizations and is also able to tile OpenCL kernels to improve their performance. Our parallelization approach is capable of automatically generating accelerated code from sequential applications, allowing users to automatically and transparently accelerate applications from diverse domains and target different accelerators without any manual effort. This combination of transparent and flexible support for different target architectures makes the RTCS unique in the domain of parallelization and offloading tools.

To demonstrate the practicality of the RTCS we evaluate the RTCS for a diverse set of benchmark applications from a broad set of domains like scientific computing, security and signal and image processing. We perform a thorough evaluation of performance gains targeting different accelerator architectures and look at the different overheads associated with the RTCS. Additionally, the RTCS is also compared against handwritten pragma-based code. Overall, our results demonstrate the practicality and the effectiveness of our approach across a broad range of application domains and accelerator architectures.

Zusammenfassung

Datenparallele Workloads im Bereich High Performance Computing begünstigen die Entwicklung von Rechnerarchitekturen in Richtung energieeffizienter heterogener Systeme. Diese Systeme sind in der Lage, geeignete rechenintensive Aufgaben auf Beschleunigern auszuführen, um den Datendurchsatz und die Energieeffizienz zu maximieren. Jedoch erhöht die Integration verschiedener Beschleuniger mit unterschiedlichen Architekturen und Speicherhierarchien in denselben Rechenknoten die Komplexität eines solchen Systems. Dies macht es schwierig, den bestmöglichen Kompromiss aus Rechenleistung und Energiebedarf zu erreichen. Die SAVE Heterogeneous System Architecture (*saveHSA*) versucht dies zu lösen, indem sie eine intelligente, selbstadaptive Schicht auf Anwendungsebene anbietet, um rechenintensive Aufgaben autonom auf die verfügbaren Ressourcen auszulagern. Dabei werden die Art der Aufgaben, Systembeschränkungen sowie die Verfügbarkeit der Ressourcen berücksichtigt, um die Ausführung bezüglich vorgegebener Ziele hin zu optimieren. Um von der saveHSA zu profitieren, müssen Anwendungsentwickler ihre Anwendung jedoch auf die jeweiligen Beschleunigerarchitekturen portieren, was nicht nur anwendungs- und domänenspezifisches Wissen, sondern auch ein tiefgreifendes Verständnis der Beschleunigerarchitekturen erfordert und letztendlich in einem erhöhten Entwicklungsaufwand und einhergehenden höheren Gesamtkosten mündet.

Um den Portierungsprozess von Anwendungen auf heterogene Systeme zu vereinfachen und den Anwendungsentwickler bei der Generierung von datenparallelem Programmcode zu unterstützen, werden nach dem aktuellem Stand der Technik verschiedene manuelle, halb- und vollautomatische Parallelisierungswerkzeuge und Compiler verwendet. Diese reichen von Pragma-basierten Ansätzen über Source-to-Source-Transformationen und Template-Bibliotheken bis hin zu neuartigen Programmiersprachen. Viele dieser Ansätze sind jedoch nicht vollständig automatisiert und erfordern vom Anwendungsentwickler ein gewisses Maß an Portierungsaufwand. Die meisten der vollautomatischen Ansätze hingegen sind nicht in der Lage, mehrere heterogene Beschleuniger zu erfassen. Unter den verschiedenen parallelen Programmiermodellen bietet OpenCL das größte Maß an Portabilität. Allerdings erfordert selbst die Entwicklung mit OpenCL einen hohen manuellen Aufwand vom Anwendungsentwickler, da bei der Parallelisierung einer Anwendung mit OpenCL neben OpenCL-Kernelcode auch entsprechender OpenCL-Hostcode geschrieben werden muss. Dieser Prozess ist zeitintensiv und fehleranfällig.

Ziel dieser Arbeit ist es, den Stand der Technik durch die Entwicklung eines neuartigen automatischen und transparenten Parallelisierungsansatzes, kurz Runtime and Just-in-Time Compilation System (RTCS), zu verbessern. Das RTCS ist in der Lage, sequenzielle Anwendungen transparent zu transformieren, um heterogene Systeme mit mehreren Beschleunigern mit Hilfe von OpenCL zu unterstützen. Die Verwendung des RTCS ermöglicht es der saveHSA, heterogene Systeme effizient zu verwalten, indem Just-in-Time (JIT) beschleunigerspezifischer Anwendungscode erzeugt wird. Das RTCS parallelisiert die Anwendung, indem es automatisch geeignete datenparallele Schleifen erkennt und in unabhängige OpenCL-Kernel umwandelt. Der entsprechende OpenCL-Hostcode, der zum Einrichten des Beschleunigers, zum Erstellen der benötigten OpenCL-Puffer, zum Übertragen der Daten sowie zum Starten des OpenCL-Kernels benötigt wird, wird ebenfalls automatisch generiert. Darüber hinaus wendet das RTCS Optimierungen bezüglich der Datentransfers an und ist auch in der Lage, OpenCL-Kernel zu kacheln (tiling), um ihre Leistung zu verbessern. Dadurch können Nutzer automatisch und transparent Anwendungen aus verschiedenen Bereichen beschleunigen lassen und ohne manuellen Portierungsaufwand verschiedene Beschleuniger verwenden. Diese Kombination aus transparenter und flexibler Unterstützung verschiedener Zielarchitekturen macht das RTCS einzigartig im Bereich der Parallelisierungswerkzeuge.

Um die Praktikabilität des RTCS zu demonstrieren, evaluieren wir es anhand einer Vielzahl von Benchmarks aus einem breiten Sprektrum von Anwendungen, wie wissenschaftliche Datenverarbeitung, Kryptographie sowie Signal- und Bildverarbeitung. Wir führen eine gründliche Bewertung der Leistungssteigerungen auf verschiedenen Beschleunigerarchitekturen durch und betrachten die mit dem RTCS verbundenen, zusätzlichen Kosten zur Kompilations- und Ausführungszeit. Zusätzlich wird das RTCS auch mit handgeschriebenem, Pragma-basiertem Programmcode verglichen. Insgesamt zeigen unsere Ergebnisse die Praktikabilität und Wirksamkeit unseres Ansatzes in einem breiten Spektrum von Anwendungsbereichen und Beschleunigerarchitekturen.

Contents

List of Tables

Listings

List of Figures

CHAPTER 1

Introduction

1.1 Motivation

Today's world has seen an increased demand for computational power. Applications such as scientific computing, data analytics, voice and gesture recognition, weather forecasting, financial analysis, medical imaging, machine learning and artificial intelligence are driving this demand. Traditionally, this demand was satisfied by building in-house data centers, and adding additional CPU nodes as the demand increased. However, building and operating a data center can be expensive and difficult to manage with aspects like cooling, power management, reliability, device lifetimes, emissions, security and resource utilization needing to be taken care of. With many users preferring not to build their own systems, the shift towards cloud computing [17] has been established, where cloud computing allows providers to profit from *economies of scale* by deploying hundreds of thousands of identical nodes.

Many applications involve a large number of arithmetic operations that can be executed concurrently. CPUs are computation devices that can handle generic computation tasks, but are inefficient when compared to dedicated hardware. On the other hand, Graphics Processing Units (GPUs) and architectures with Many Integrated Cores (MICs) are designed to perform massively parallel computations, offering a considerable advantage in performance and energy efficiency when compared to CPU-only systems. Such data parallel workloads are driving architectures toward efficient, low power heterogeneous systems with CPUs, GPUs and MICs physically integrated into a single compute node. Such heterogeneous systems are able to improve the overall performance of the system by executing control-intensive code on a general-purpose host CPU and executing computationally intensive data-parallel kernels on accelerators designed to maximize data throughput.

Many companies (like Amazon EC2, Google Cloud Platform, IBM SoftLayer, Microsoft Azure, Rackspace, etc.) provide on demand access to scalable, heterogeneous nodes featuring state-of-the-art multi-core CPUs, GPUs or MICs (Intel Xeon PHI). Such systems are capable of processing parallel workloads very efficiently, while being more energy efficient than regular CPU-only clusters. For many concrete workloads, regular CPU systems combined with accelerators can reach higher levels of performance whilst being more energy efficient [8]. Performance and energy efficiency requirements are the main driving force behind the popularity of such accelerators in High Performance Compute (HPC) centers, where applications incorporating several types of algorithms are able to benefit from the distinct architectures of each accelerator [10].

Integrating different accelerators from different vendors, consisting of different architectures and memory hierarchies into the same compute node, brings its own set of challenges. In an effort to standardize platform design and in order to unlock the performance and power efficiency of modern parallel accelerators, the HSA Foundation [39] has developed the Heterogeneous System Architecture (HSA). The HSA introduces multi-vendor architecture support allowing heterogeneous execution on different resources like CPUs, GPUs, DSPs, FPGAs and even on complex systems-on-chip (SoCs), bringing heterogeneous computing to platforms like mobile devices, desktops and high-performance computing (HPC) systems [40]. However, this increase in heterogeneity also increases the complexity of such systems.

To be able to achieve optimal performance/energy trade-off on such systems, system architects need to take into account the application mix, their workload, as well as the efficiency and utilization of different heterogeneous resources. The SAVE project (Self-Adaptive Virtualization-aware high-performance/low-Energy heterogeneous system architectures) [23] tries to address these limitations by extending the HSA and introducing self-adaptiveness and hardware-assisted virtualization for dynamically changing working scenarios. This state-of-the-art architecture is known as the *saveHSA* and allows the system to autonomously decide which computing resources are exploited to achieve efficient execution based on user-defined optimization goals like performance or energy goals [72]. It achieves this by adding a smart self-adaptive layer to the operating system that is responsible for distributing tasks to different heterogeneous resources (CPUs, GPUs, MICs, etc.).

This advanced run-time and self-adaptive layer is known as the Orchestrator [13] and enables the SAVE heterogeneous system to efficiently manage accelerators for diverse applications with dynamic workloads while at the same time satisfying application level and overall system level goals. The Orchestrator is capable of dynamically and seamlessly offloading the workload to available resources by taking into account the changeable workloads, application goals, system constraints and resource availability. However, this is only possible when the corresponding accelerator code is available. This means that application developers need to port their applications to different accelerators, to fully benefit from the Orchestrator and the *saveHSA*. Accelerators in heterogeneous systems are diverse, with different architectures, programming techniques, tools, compilers and optimizations. Application developers not only require application and domain-specific knowledge, but also need to understand the different target accelerators, leading to an increase in the design effort and costs.

Developers might choose to use one of the different standards, pragmas, templates, extensions, languages or various semi and fully automatic parallelization compilers and tools that simplify the task of porting sequential applications to parallel accelerators. However, many of these approaches often support just a single architecture and are only capable of targeting a subset of the accelerators in heterogeneous systems. Among the programming models that expose many platform details to the developer, from a functional perspective, OpenCL [74] is the most portable one, supported by multi and manycore CPUs, GPUs and recently also by FPGAs [73, 43]. However, OpenCL not only poses the challenge of extracting hotspots into kernels and optimizing them for the target accelerator architecture, it also involves writing OpenCL host code to initialize OpenCL devices, create buffers on the accelerator, transfer the required data to the accelerator and finally launch the kernel on the accelerator. This is a time consuming and error-prone process. Given these challenges, there is a considerable gap between the architectural potential of highly heterogeneous multi-accelerator architectures and their actual adoption and utilization.

1.2 Contributions of this Thesis

The main contribution of this thesis is the development of a novel approach that can automatically parallelize sequential applications and transparently generate accelerated code targeting different heterogeneous multi-accelerator architectures. Our approach enables users to automatically and transparently accelerate applications from diverse domains and offload computation to different accelerators without any manual effort. Our approach partially builds upon and integrates results from different open-source projects like LLVM [53] and Axtor [65]. A fork of our approach known as *HTrOP* has been released as an open-source project and is publicly available under the MIT license on GitHub [41].

The main contributions of this thesis are:

1. We present a novel Runtime and Just-in-Time Compilation System, known as the RTCS, which is a user-transparent end-to-end approach and practical realization for bringing sequential programs to parallel execution on different accelerator targets via OpenCL.

2. Based on the analysis of loop structures in applications, we identify parallelization opportunities and present a technique that automatically detects and transforms suitable data-parallel loops into independent OpenCL-typical work-items that are executed in parallel (OpenCL kernels).

3. Not only do we generate parallelized OpenCL kernel code, we also automatically generate the intricate OpenCL host code required to setup the OpenCL device, create OpenCL buffers, transfer data to the device and launch the OpenCL kernel on the device.

4. Focusing on improving the performance of the generated OpenCL kernels, we present an automatic tiling approach that exploits shared memory to improve global memory efficiency.

5. We further improve application performance by analyzing the application and applying data transfer optimizations to reduce the amount of data transferred to and from the accelerator.

6. Integrating our transparent and automated approach into the self-adaptive *saveHSA* allows the *saveHSA* to efficiently manage heterogeneous systems by using our RTCS approach to JIT generate accelerator specific code that was previously unavailable.

7. In the domain of parallelization and offloading tools, the RTCS is unique in its combination of transparency and flexible support for different target architectures.

8. Since we make use of different open-source projects in our approach, we also give back to the open-source community by releasing a version of our approach as an open-source project on GitHub.

We also perform a thorough evaluation of performance gains targeting different accelerator architectures, with a comparison against handwritten pragma-based code. Overall, our results demonstrate the feasibility and the effectiveness of our approach by improving the overall performance for a wide range of application domains and different accelerator technologies. The *saveHSA* in turn is able to benefit by improving the system's performance, utilization and energy efficiency. The RTCS can achieve comparable performance to handwritten OpenACC code, while being fully automated. A particular quality of our OpenCL-based approach is that it treats additional CPU cores beyond the first one essentially like any discrete accelerator (mCPU), which can greatly facilitate resource allocation and utilization in a dynamic, heterogeneous HPC context.

The SAVE Project: A large portion of this work was performed as part of the SAVE project [23]. SAVE stands for "Self-Adaptive Virtualization-aware high-performance/low-Energy heterogeneous system architectures" and was a collaborative project between leading partners in the field of academic research and industry under the European Union Seventh Framework Programme (FP7). The consortium consisted of Politecnico di Milano, Technological Educational Institute of Crete and Paderborn University on the academic side, while STMicroelectronics, Virtual Open Systems, Maxeler Technologies and ARM made up the industry part of the consortium.

The aim of the SAVE project was to develop state-of-the-art hardware and software technologies that are able to efficiently exploit heterogeneous system architectures. Politecnico di Milano and Paderborn University focused on the software aspect, while the remaining members of the consortium looked at the hardware aspects (e.g. virtualization, cache coherency, etc.). Politecnico di Milano's main area of research was the self-adaptive system (Orchestrator), while we explored different techniques to automatically parallelize sequential code and transparently offload computation to different types of accelerators in heterogeneous systems (RTCS).

To this extent, we have presented our collaborative work and published peer-reviewed results in distinguished conferences and journals. We have also received a best paper award for [Vaz14] with Gavin Vaz being the first author. The author's publications are summarized before the main bibliography on page 97.

1.3 Thesis Structure

The remainder of this thesis is structured as follows:

Chapter 2 looks at the different state-of-the-art programming languages and extensions that can be used to reduce the complexities of accelerator programming. Different acceleration techniques that try to automatically accelerate legacy applications are also explored.

Chapter 3 describes the saveHSA-Orchestrator in detail and looks at its inner workings. It also introduces the SAVE-Enabled application and describes its execution.

Chapter 4 gives a brief introduction to the LLVM compiler infrastructure and the OpenCL framework, both of which are required to understand our approach.

Chapter 5 is devoted to the main contributions of this thesis and presents our Runtime and Just-in-Time Compilation System approach in detail.

Chapter 6 presents a detailed evaluation of our approach and includes the evaluation of the overheads as well as the performance. Additionally, our approach is also evaluated against handwritten pragma-based code.

Chapter 7 summarizes the work presented in this thesis and discusses directions for future research.

CHAPTER 2

Related Work

Modern heterogeneous systems have multiple accelerators attached to the same compute node that offer different levels of parallelism. Such systems are capable of providing high performance with low energy consumption. However, legacy applications need to be first ported to the accelerators in order to take advantage of their performance. The task of parallelizing and porting legacy code to heterogeneous systems lying squarely on the shoulders of the application developer. This requires the understanding of the application, determining the data and control dependencies and then writing the parallel code for different accelerators that need to be supported. Differences in the accelerator architectures, their memory hierarchies and level of parallelism makes this a very challenging and daunting task.

In this chapter, we first look at how new programming languages or extensions can be used to make accelerator programming easier. Secondly, we look at different acceleration techniques that try to automatically accelerate legacy applications.

2.1 Accelerator Programming

In the manual approach, the developer has to identify the appropriate regions of the application that might benefit from parallel execution on an accelerator. This analysis could either be performed by visual inspection of the code by an experienced developer or by using profiling tools to determine application hotspots (or bottlenecks). The identified hotspots need to be then manually adapted for the particular accelerator or programming model. This is a time consuming and error-prone process, as each model has its own APIs and level of abstraction.

Developers can choose to accelerate their applications by using pragma-based language constructs (when supported) or write accelerator specific code from scratch. Pragma-based approaches make use of compiler directives to guide the compiler. Pragmas can be ignored by non-conforming compilers or runtime environments. Thus, when code with pragmas meets the right environment, it is executed with acceleration, otherwise without. APIs such as OpenMP [21] and OpenACC [34] use pragmas to annotate regions that should be accelerated. Instead of writing accelerator code from scratch, the developer only needs to insert pragmas at the right places in the application code and the compiler does the heavy lifting by taking care of the thread and data management, freeing the developer from low-level details.

Threading Building Blocks (TBB) [50] on the other hand, supports parallel programming with the help of a C++ template library. TBB makes it easier to write parallelized applications when compared to other threaded approaches. However, the application developer needs to completely rewrite application hotspots to be able to take advantage of TBB. Additionally, at the moment, TBB can only target multi-core processors, making it unattractive for use in heterogeneous systems.

GPUs are one of the most prominent accelerators being currently used in High Performance Computing (HPC). The CUDA [66] programming language was the first programming language to give the developer precise control of the accelerator and direct access to the GPU's parallel computational elements. Developers can use CUDA to accelerate C/C++ or Fortran applications by modifying the application to execute computationally intensive parts of the application on the GPU. CUDA provides extensions for C/C++ and Fortran with additional qualifiers and keywords that allow the developer to program the GPU. It also provides different Application Programming Interfaces (APIs) and compiler directives to manage and launch kernels on the GPU. CUDA is however a proprietary programming language created by Nvidia and can only be used to program Nvidia GPUs.

C++ Accelerated Massive Parallelism (C++ AMP) is a programming model which unlike CUDA works on GPUs from different vendors. C++ AMP is an open specification [19] and is capable of compiling applications to execute on data-parallel accelerators. It allows the programmer to express parallelism directly in C++. Since C++ AMP is an open specification, different compiler toolchains can choose to implement it to support different accelerators. C++ AMP is published by Microsoft and is a part of their Visual Studio IDE and is currently only supported on Windows machines [84].

OpenCL [74] provides an open standard interface for parallel computing using task and data-based parallelism, which can be executed across different heterogeneous devices. Unlike CUDA, C++ AMP and TBB, OpenCL is not restricted to a specific platform or to a specific vendor. OpenCL is backed by a large number of hardware vendors like Intel, IBM, ARM, Xilinx, AMD, Nvidia and Qualcomm to name a few. There exist several conformal implementations of OpenCL that support multi-core CPUs, GPUs, MICs (Intel Xeon PHIs [46]), DSPs and more-recently FPGAs [73, 43]. The programming model is similar to CUDA where the developer needs to explicitly express parallelism. OpenCL specifies unified APIs to control the device, transfer and handle the data and execute the hotspots (see Section 4.2). The OpenCL standard abstracts away the different accelerators and their architectures as *OpenCL devices*, allowing the developer to target diverse accelerator architectures with the same OpenCL kernel code. This functional portability, however does not guarantee performance portability across different accelerator architectures (see Chapter 6).

2.2 Transparent Acceleration and Parallelization

Transparent acceleration aims to parallelize sequential applications by automatically detecting application hotspots and generating corresponding accelerated code. Commercial compilers like the Intel C++ Compiler are able to automatically translate sequential parts of an application into equivalent multi-threaded code. The compiler analyses the code to determine loops that contain parallelism and are good candidates to parallelize. It also performs dataflow analysis and partitions the application for threaded code generation [44]. The compiler then transforms the parallelizable loops into separate threads that can be executed in parallel. The Intel compiler can target both Intel and non-Intel microprocessors as well as the Intel Many Integrated Core Architecture. However, automatic parallelization for GPUs and other heterogeneous resources are presently not available.

The SUIF (Stanford University Intermediate Format) compiler system [1, 57] is capable of automatically translating sequential dense matrix code into parallel code for large-scale parallel machines. The compiler first accepts sequential C or FORTRAN code and translates it into an intermediate representation. The code is then analyzed for parallelism and data locality after which it is optimized and finally the parallel code is generated. However, the generated accelerated code can only target shared address space machines like the Stanford DASH multiprocessor[54] and the Kendall Square Research KSR-1.

Tournavitis et al. [78] developed a semi-automatic approach that uses profiling-based analysis along with machine-learning to parallelize sequential applications. Dynamic profiling data is used to determine the data and control dependencies of the application. Based on this analysis, the source code is annotated with OpenMP pragmas for parallel loops. A previously trained machine-learning based prediction mechanism is used to decide if and how a parallel loop candidate should be parallelized. In this step, if the predictor determines that the loop will benefit from parallelization, OpenMP work allocation clauses are added to the code, otherwise the OpenMP annotations are removed. However, the user is required to approve the loops that were parallelized automatically making it a semi-automatic approach.

Cetus [22] is a Java based automatic parallelization tool that is capable of performing source-to-source transformation of C applications. Cetus analyses the application for data dependencies with Banerjee-Wolfe inequalities [6] and also detects reduction-variables. It then parallelizes the code by performing array and scalar privatization and induction-variable substitution. Cetus however only supports multi-core accelerators. Other auto-parallelizing compilers like ROSE [56], Polaris [12], or auto-parallelization approaches like [9], or auto-parallelization tools like the ParaWise Expert Assistant [47] and S2P [4] only target shared memory multi-core processor directly or via OpenMP and are unable to target the different accelerators of a heterogeneous system.

The PGI [69] compiler is capable of generating a heterogeneous executable by combining different accelerator implementations into a single executable. It also supports auto-parallelization for Fortran, C and C++ applications. The compiler searches for and then automatically parallelizes loops that do not contain any cross-iteration data dependencies. It ignores loops with calls to other functions as well as loops with low iteration counts. However, compiler switches can be used to override most of these restrictions. Presently, the PGI compiler only supports multi-core processors and Nvidia GPUs.

The Par4All [68] auto-parallelizing and optimizing compiler was a source-to-source compiler aimed to merge various open-source developments in order to transform sequential applications into parallel ones. It accelerated C and Fortran sequential programs by generating parallelized sources for different hardware platforms like OpenMP, CUDA and OpenCL. Vendor specific tools and compilers where then used to generate the accelerator specific executable for multi-core systems and other parallel accelerators like GPUs. Par4All looked like a promising tool that could target heterogenous accelerators, however the development efforts were frozen after SILKAN withdrew their support in 2012.

PLuTo [15] is an end-to-end framework that is capable of automatically parallelizing and optimizing sequential applications. Internally PLuTo makes use of the polyhedral model [32] to reorder the loop execution with improved cache locality or to parallelize them. First, a polyhedral model is used to perform static dependence analysis. The loop is then transformed to expose parallelism and improve data locality by performing tiling or fusing the loops by using affine transformations. Finally, accelerated (annotated) OpenMP code is generated using CLooG [7]. CLooG stands for *Chunky Loop Generator* and was designed to be the back-end of the polyhedral model. Similar to previous tools, it can only target accelerators that are supported by OpenMP.

Grosser et al. [33] also make use of the polyhedral model in their Polly-ACC compiler. Polly-ACC is a heterogeneous compiler capable of automatically parallelizing applications. Like the PGI compiler, it can also generate hybrid executables (CPU + GPU). Hotspots are identified by detecting Static Control Parts (SCoPs) of an application. A modified version of the PLuTo scheduler [14] is used to expose parallelism and increase data locality. PPCG [80] is used to generate a profitable accelerator schedule for GPUs while LLVM's NVPTX [79] back-end is used to generate code for Nvidia GPUs. On the other hand, Kalms et al. [48] are able to generate OpenCL code by using the polyhedral model. Similar to Polly-ACC, their source-to-source compilation framework also works on SCoPs. The main focus of their work is the `PPCGSourceCodeGeneration` module that converts SCoPs into the PPCG format and then generates OpenCL host and device code via a custom LLVM Intermediate Representation (LLVM IR) to C code writer. Although their approach is able to target OpenCL, it cannot generate accelerator optimized OpenCL code and lacks hybrid execution with the host code being able to only offload to a single accelerator.

Damschen et al. [Dam15] describe a LLVM-based client-server architecture called BAAR where hotspots are also detected as SCoPs and offloaded at runtime to an Intel Xeon PHI. In the envisioned architecture, the client machine only consists of a low-power CPU and the server has the PHI attached to it (loosely coupled). To accelerate a SCoP, the client issues a remote procedure call (RPC) with the LLVM bitcode and sends the function arguments via MPI [31] to the server. The server generates code and executes the request. To utilize the vector processing units and manycore processors of the PHI, the code is vectorized using Loop and Superword-Level Parallelism (SLP) provided by LLVM [5]. The sequential code is parallelized with OpenMP to generate sufficient computing threads. However, this approach has a number of drawbacks: using SCoP limits the loops that can be parallelized; targeting OpenMP instead of OpenCL limits this approach to a smaller number of heterogeneous devices; the server can easily become a bottleneck for multiple clients and the RPC calls cause additional data transfer overheads which need to be amortized. In our work, the execution environment is tightly coupled to the accelerators and the OpenCL kernel code is generated by a centralized service.

2.3 Chapter Conclusion

In this chapter, we looked at the different standards, templates, extensions and languages that developers can exploit to simplify the task of porting sequential applications to parallel accelerators. However, most of these approaches are only capable of targeting multi-core processors, manycore processors or Nvidia GPUs. Additionally, some tools like C++ AMP only work on specific platforms. OpenCL on the other hand supports diverse accelerators, making it highly relevant in the HPC context, but also particularly interesting for mobile devices and embedded platforms [29, 55, 30]. The only caveat of using OpenCL is that one needs to manually translate the application. This involves writing OpenCL kernel code for the application hotspots, as well as writing OpenCL host code to transfer data and launch the kernel on the accelerator. This is a time consuming and error-prone process. In our approach, we automatically identify application hotspots and generate OpenCL kernels which are then automatically integrated into the application by generating the corresponding OpenCL host code.

Table 2.1: Parallelization approaches, required developer activity and target architectures.

	OpenMP	OpenACC	OpenCL	others	RTCS
parallelization	pragmas	pragmas	kernel + host code	automatic	automatic
multi-core CPU	supported	supported	supported	Polly [32] (OpenMP)	demonstrated
Xeon Phi	supported	-	supported	-	demonstrated
Nvidia GPU	experimental	supported	supported	PollyACC [33] (NVPTX)	demonstrated
other GPUs	experimental	-	supported	-	possible
FPGAs	research	research	supported	-	research

We also looked at various semi and fully-transparent parallelization compilers and tools. These approaches are capable of targeting specific accelerators, either by direct compilation to parallel architectures or by employing source-to-source transformations to target languages like OpenMP that support compilation to parallel accelerators. Most of these approaches either target multi-core processors, manycore processors or Nvidia GPUs and are not suitable for the heterogeneous system containing other types of accelerators. In our work, we try to bridge this gap by developing a fully automated and transparent compilation and runtime approach that allows sequential legacy programs to be executed in parallel on different heterogeneous accelerator targets via OpenCL. Table 2.1 summarizes and compares the most relevant alternative approaches with the RTCS.

In the next chapter, we will take a look at the Orchestrator in detail and how it uses application-level as well as system-level goals to select the optimal accelerator for a given application.

CHAPTER 3

The saveHSA-Orchestrator

The SAVE heterogeneous system is able to efficiently manage accelerators for diverse applications with dynamic workloads. The Orchestrator is an advanced run-time and self-adaptive operating system layer that is the primary component of the *saveHSA*. The Orchestrator is developed by Bolchini et al. [13] and is capable of dynamically and seamlessly offloading the workload to available resources by taking into account the changeable workloads, application goals, system constraints and resource availability. It is responsible for balancing the different accelerator resources to improve their overall performance, utilization, dependability, programmability and self-awareness in HPC systems.

The Orchestrator makes use of the Heartbeat framework [38] to dynamically monitor the performance of the executing applications. Each application sends the Orchestrator periodic heartbeats via a Heartbeat API. These heartbeats are used by the Orchestrator to monitor the quality of service (QoS) or performance of the application. The QoS is represented as the number of heartbeats per second and is known as the *heart-rate*. For example, in a video processing application, a heartbeat is issued after processing each frame, making the heart-rate equivalent to the frames per second. The application developer can also use the Heartbeat framework to set application level goals. In the case of the previous video processing example, a minimum and/or maximum number of frames per second can be set as the application's goal.

System managers can also set system level constraints for the Orchestrator. The Orchestrator monitors the performance of the applications (heartbeats) as well as the status of the system and its resources. It uses this information to determine the optimal resource for a given application such that application-level goals as well as system-level constraints are met and then offloads the computation to the selected resource. These types of feedback loops are referred to as Observe-Decide-Act (ODA) loops and are one of the essential building blocks in self-aware and adaptive systems. An overview of the Orchestrator along with its interaction with the applications and HPC resources is depicted in Figure 3.1. The CPU represents a single-core CPU, the mCPU represents a multi-core CPU while the

GPGPU represents a General-Purpose GPU. In the next section, we take a closer look at the Observe-Decide-Act loop.

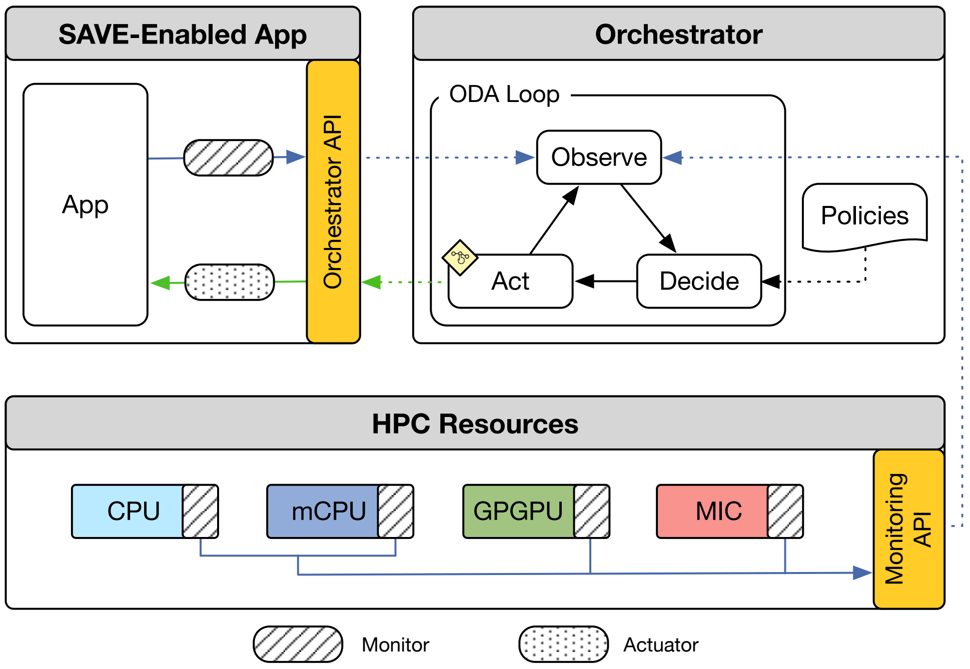


Figure 3.1: Overview of the *saveHSA* and the Orchestrator.

3.1 The Observe-Decide-Act Loop

The Orchestrator is able to adapt application as well as system behavior of the system by making use of the ODA loop. As indicated by its name, the Observe-Decide-Act loop is comprised of the following three phases:

- In the **Observe** phase, the Orchestrator monitors the different properties of the applications as well as the different heterogeneous resources of the HPC system. In Figure 3.1, the slanted lines show us where the Orchestrator monitors the application and the system, while the blue lines represent the flow of the monitoring data to the Orchestrator. The precise type of data collected depends on the concrete optimization objectives of the ODA loop. Table 3.1 summarizes what data the Orchestrator can collect during the observe phase of the ODA loop according to Bolchini et al. [13].

  Applications can communicate their goals and constraints to the Orchestrator via an API. They can also register different accelerator implementations with the Orchestrator. For example, in addition to a CPU implementation, an application developer might choose to also provide accelerated code targeting a GPU or a MIC. Each accelerator implementation is known as an actuator and allows the Orchestrator to

Table 3.1: Monitoring information that can be collected by the Orchestrator [13].

Application Status	Information on performance and energy consumption monitored for each application	Throughput Latency Deadline Performance per Watt Energy consumption
Resource Status	Information on hardware resources available through the Monitoring Interface	Power Energy Temperature Utilization Frequency & Voltage

dynamically modify the behavior of the application at runtime by choosing a different accelerator implementation.

Additionally, system managers can also specify system-level goals and constraints for the Orchestrator. Table 3.2 summarizes the different application and system-level goals and constraints supported by the Orchestrator according to Bolchini et al. [13]. Depending on the application mix, system managers can also devise different types of policies to control application as well as the overall system behavior (e.g. an *earliest deadline first* policy or a *throughput-oriented heuristic*).

- The Orchestrator has a global overview of the system and is aware of all the executing applications, their goals, constraints and also the different actuators that are available. In the **Decide** phase, the Orchestrator uses this information and applies policies to the data collected in the observe phase to determine the optimal resource to dispatch the application to such that application and system-level goals and constraints are satisfied.

- In the **Act** phase, the Orchestrator instructs the application to execute on the selected resource by actuating the appropriate actuator and is illustrated by the green lines in Figure 3.1.

In this section, we saw that in order for the Orchestrator to dynamically modify the behavior of an application at runtime, the application needs to support the heartbeat framework as well as provide the Orchestrator with a mechanism to modify its behavior at runtime. Applications that support this are known as SAVE-Enabled applications and in the next section, we take a closer look at them.

Table 3.2: Various application and system-level goals and constraints supported by the Orchestrator [13].

Application-level	Constraints (i.e., setting minimum/maximum thresholds that should not be exceeded)	Throughput Latency Deadline Performance per Watt Energy consumption
	Goals (i.e., metrics to be optimized)	Throughput Latency Performance per Watt Energy consumption
System-level	Constraints	Power Energy consumption
	Goals	Power Energy consumption

3.2 SAVE-Enabled applications

For an application to benefit from the self-adaptive *saveHSA*, the application should:

- provide multiple accelerator specific implementations (actuators)
- use the Orchestrator API
- be able to report its progress via heartbeats

Heterogeneous HPC nodes have different accelerators with different computational properties. These properties can be exploited by the Orchestrator by modifying the behavior of an application/system at runtime. The Orchestrator achieves this by dynamically offloading the computationally intensive part of the application (hotspots) to the most suitable resource. However, this is only possible if the application provides and registers different accelerator specific hotspot implementations (actuators) with the Orchestrator. Application developers need to create different accelerator specific implementations for the application hotspot and then register them with the Orchestrator using the Orchestrator API. This API is also used to set the application-level goals (in terms of throughput, latency, etc.).

Applications that implement the Orchestrator API are able to benefit from the *saveHSA* and are referred to as SAVE-Enabled applications (SEAs). The typical structure of a SEA and its interaction with the Orchestrator is depicted in Figure 3.2. Initially, the application is initialized and the application-level goals are set using the Orchestrator API. We can see that in this example, the application implements a CPU as well as a GPGPU version of the hotspot, which are then registered with the Orchestrator via the Orchestrator API.

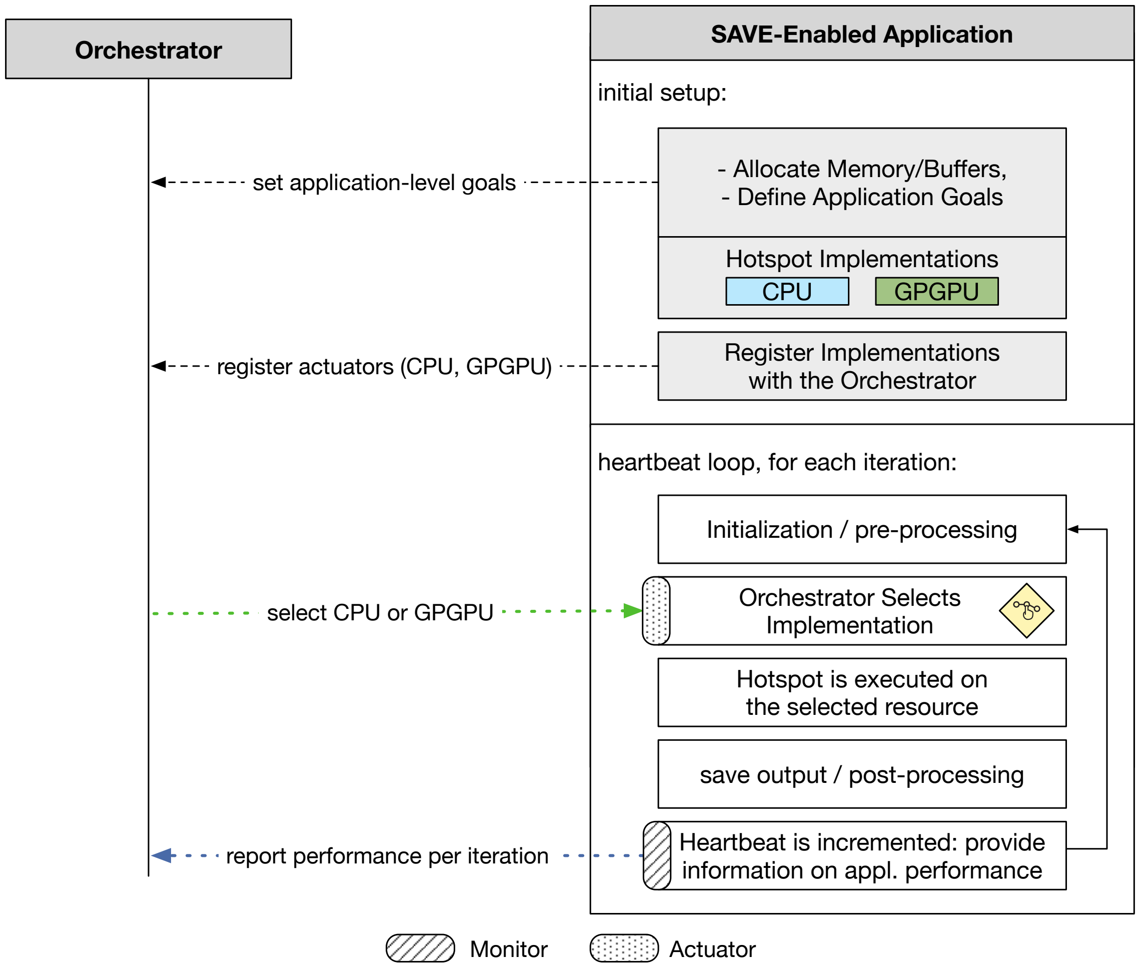

Orchestrator
SAVE-Enabled Application
initial setup:
set application-level goals
- Allocate Memory/Buffers,
- Define Application Goals
Hotspot Implementations
CPU
GPGPU
register actuators (CPU, GPGPU)
Register Implementations with the Orchestrator
heartbeat loop, for each iteration:
Initialization / pre-processing
select CPU or GPGPU
Orchestrator Selects Implementation
Hotspot is executed on the selected resource
save output / post-processing
report performance per iteration
Heartbeat is incremented: provide information on appl. performance
Monitor
Actuator

Figure 3.2: Structure of a SAVE-Enabled application and its interaction with the Orchestrator.

The Orchestrator needs to monitor the application in order to determine the QoS/performance of the application. This is represented by the heartbeat loop in Figure 3.2. It is known as the heartbeat loop because after each loop iteration, the application's execution progress is reported to the Orchestrator in the form of a Heartbeat. Additionally, the Orchestrator also needs to instruct the application to execute on the resource it chose during the Decide phase. This is done at the beginning of each iteration in the heartbeat loop, before the hotspot is executed. If an application is not a SEA, it will still be executed on the *saveHSA*, but it will not profit from the advanced *saveHSA* features. To help understand how the Orchestrator interacts with the heartbeat loop, we briefly look at how a heartbeat loop is executed with the help of an example.

Execution of a Heartbeat Loop The typical execution of a heartbeat loop is depicted in Figure 3.3. At the beginning of the heartbeat loop, the Orchestrator has the possibility of switching between the implementations that have been previously been defined by the application programmer and registered with the Orchestrator. In this example, the platform supports four resources (CPU, mCPU, GPGPU and MIC), however only two implementations of the hotspot have been provided (CPU and GPGPU). Allowing the Orchestrator to adapt the performance/behaviour of the application by selecting either the CPU or GPGPU at the beginning of each loop iteration. Even though the platform has a MIC available, the Orchestrator cannot exploit it as the application does not provide an implementation for the MIC.

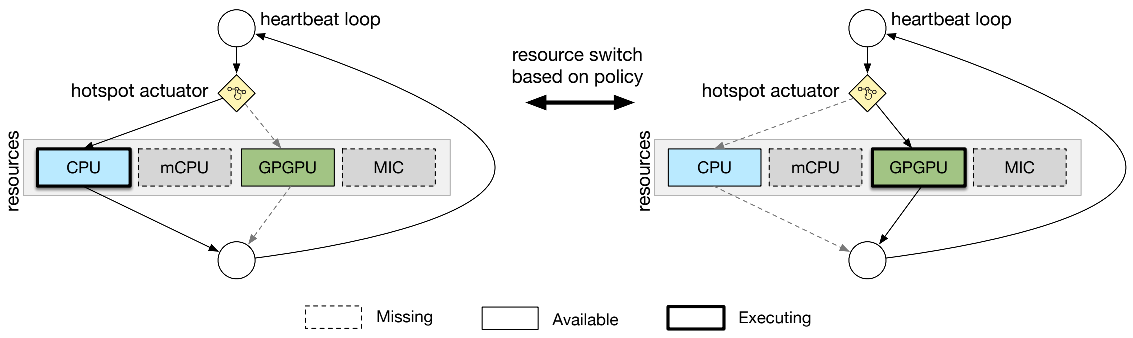


Figure 3.3: Heartbeat loop of a SAVE-Enabled application.

3.3 Chapter Conclusion

In this chapter, we looked at the inner working of the Orchestrator and how it plays an integral part of the *saveHSA*. We have seen that the Orchestrator can transform current static workload-based systems to systems that are self-aware and capable of running real-world applications with on-demand computation requirements for dynamic workloads and different performance requirements. While at the same time balancing the use of accelerator resources to improve performance, utilization, dependability and overall energy utilization of such systems.

However, the Orchestrator is only able to achieve this if different accelerator specific implementations for the application hotspot are available. This forces application developers to port their applications to different accelerators in order to benefit from the *saveHSA*. Accelerators in heterogeneous systems are diverse, with different architectures requiring different programming techniques, tools, compilers. Application developers not only require application and domain-specific knowledge, but also need to understand the different target accelerators in detail, leading to an increase in the design effort and costs. In the following chapters we introduce our Runtime and Just-in-Time Compilation System that attempts to bridge this gap.

CHAPTER 4

Background

The Runtime and Just-in-Time Compilation System builds upon the LLVM compiler infrastructure to perform code analysis, parallelize hotspots, generate accelerate code and integrate the newly generated code into the already executing application. Additionally, the RTCS makes use of OpenCL to target multiple accelerators. Hence, in this chapter, we take a look at the LLVM compiler infrastructure as well as the OpenCL framework.

4.1 LLVM Compiler Infrastructure

The LLVM compiler infrastructure is an open-source project comprising of a "collection of modular and reusable compiler and toolchain technologies" [60, 53]. The LLVM compiler employs a three-phase compiler design to compile an application from source code to binary code. Figure 4.1 shows the different phases of the LLVM compiler.

The *front-end* is responsible for parsing the source code, diagnosing errors and translating it into an intermediate representation (LLVM IR). There are many front ends available, each supporting different high-level programming languages. *clang* is the most popular one, and is used for C/C++ programs. *llgo*, *flang* and *rust* are a few of the other front-ends and support Golang, Fortran and Rust respectively. Similar to traditional compilers, the LLVM front-end includes lexical analysis, syntactic analysis, semantic analysis and intermediate code generation phases. The LLVM IR is a low-level programming language representation that is similar to assembly instructions which allows high-level programs to be represented in a machine independent form. The LLVM IR is represented in a static single assignment (SSA) form where all variables need to be defined before they are used and each variable is assigned exactly once.

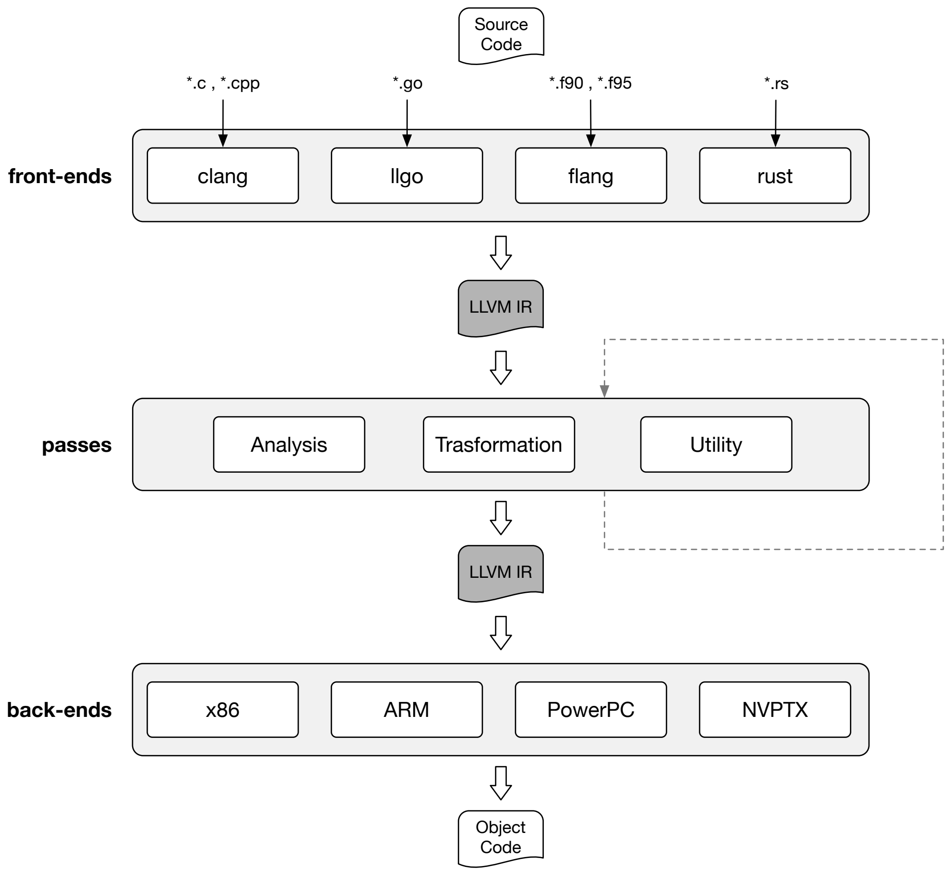


Figure 4.1: Phases of the LLVM compiler toolchain.

One of the main advantages of LLVM over traditional compilers is that the LLVM IR can be analyzed, optimized and transformed using different LLVM Passes.

- The *Analysis Passes* analyze the LLVM IR and generate data structures that other passes can access and use. Generally, analysis passes do not modify the LLVM IR. Alias analysis, dom tree construction or basic call graph construction are a few examples of analysis passes.

- *Transformation Passes* on the other hand, modify the LLVM IR in some way or the other. These passes are generally used to perform code optimizations. Dead code elimination, constant propagation or combining of redundant instructions are a few of the available transformation passes.

- The *Utility pass* provides some bookkeeping or utilitarian functionality such as a pass for viewing the CFG of a function, which cannot be classified as analysis or transformation pass.

All the available passes are summarized in the LLVM documentation [62]. Multiple passes can be run in sequence to optimize the LLVM IR with the help of the LLVM Pass Manager which optimizes their execution by avoiding re-computation of analysis results as much as possible and scheduling the passes to run efficiently by pipe-lining them together. In our approach, we make use of inbuilt LLVM analyses and transformation passes as well as write our own custom passes to analyze the application and parallelize the hotspots.

The LLVM *back-ends* use the LLVM IR to generate target specific machine code. LLVM supports a variety of target instruction sets, including x86, x86-64, ARM, PowerPC or the Nvidia Parallel Thread Execution (NVPTX) to name a few.

The notable feature of this three-phase compartmentalized LLVM compiler infrastructure is that it is relatively easy for a compiler developer to add new components to a particular phase. For example, adding a new code optimization requires the developer to write a new analysis and/or transformation LLVM pass [83] without being concerned about the front-end or back-end. Similarly, targeting a new architecture requires little or no understanding of the front-end or passes modules.

In addition to the traditional compilation work-flow that creates a machine executable binary for a specific target, the LLVM compiler infrastructure also provides us with an execution environment similar to the Java VM and is known as the *Execution Engine*. It supports an interpretation-based execution model as well as a JIT execution model (Figure 4.2). The interpreter translates the program one statement at a time while the Machine Code JIT (MCJIT) execution engine compiles the entire module before executing any function. It does not support lazy function-level compilation, but has broader platform support and better tool integration. Latest versions of the LLVM supports the On-Request Compilation MCJIT (ORC MCJIT) execution engine which is a modular implementation of the MCJIT's design philosophy of relying on the MC layer while supporting the lazy compilation of functions. In our approach, we use LLVM 3.8.0 and make use of a modified version of the MCJIT that allows us to recompile and link functions at runtime.

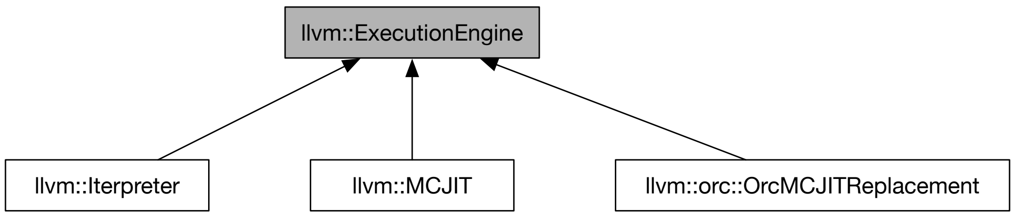


Figure 4.2: The inheritance diagram for the Execution Engine [61].

Binary applications in the form of non-annotated LLVM Intermediate Representation are used as the input to our runtime approach. This not only allows us to target applications that can be compiled into, or distributed in LLVM Intermediate Representation, but also legacy applications (machine code) which can be decompiled [3] into LLVM IR. This enables us to apply our approach in scenarios where the application source code is not available [36]. In our approach, we leverage the LLVM compiler infrastructure to generate and integrate accelerator specific code. The application is analyzed and transformed via.

custom analysis, optimization and transformation passes. Additionally, the LLVM MCJIT execution engine is used to JIT compile and integrate the newly generated code into the already executing application at runtime.

4.2 OpenCL Framework

According to Khronos, OpenCL (Open Computing Language) is an open, royalty-free standard for cross-platform, parallel programming of diverse accelerators [77]. The goal of the RTCS is to transparently accelerate sequential applications by offloading computationally intensive parts of the application to heterogeneous accelerators. The OpenCL standard abstracts different accelerators and their architectures as *OpenCL devices* allowing us to target diverse accelerator architectures with the same OpenCL code. Figure 4.3 shows an overview of the heterogeneous OpenCL architecture comprising of the *Host* and the connected OpenCL devices. Each OpenCL device consists of a large number of Processing Elements which are grouped together into different Compute Units. The number of processing elements and compute units for a particular device is determined by the device vendor. All processing elements belonging to a single compute unit execute the same instruction in lockstep but work on different data.

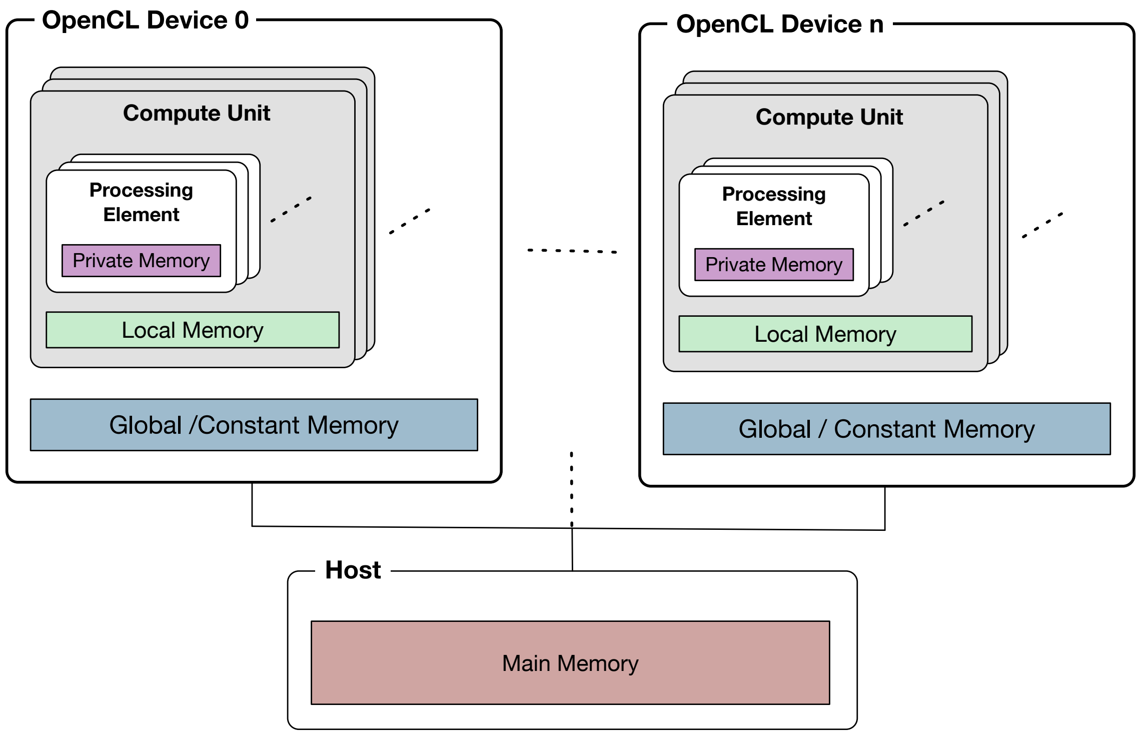


Figure 4.3: OpenCL platform model.

The host itself does not perform any OpenCL computation, but uses the OpenCL API to manage the OpenCL devices, transfer the required data from main memory to the device (and back), and call the required OpenCL kernels with the appropriate parameters. The kernels can be compiled at run-time making them portable across different accelerators (OpenCL devices). The *kernel* is a parallelized OpenCL C function (hotspot) that can be executed independently and in parallel on an OpenCL device. Each instance of a kernel execution is known as a *work-item* which executes the same kernel code but works on different data. The iteration space of the kernel is represented as a N-Dimensional grid (ND-Range), and is used to call the kernel from the host.

Figure 4.4 illustrates how a 2-dimensional ND-Range (work items) is divided into multiple work-groups. This is either done automatically by the OpenCL runtime or can be manually specified by the application programmer. All work-items belonging to a work-group execute on the same compute unit which allows the work-items to share (fast) local memory of the compute unit enabling them to synchronize with one another. However, depending on the application, such kernels should use synchronization points (barriers) to guarantee that all the work-items belonging to the same work-group have reached the synchronization point before executing the next statement.

Internally, OpenCL divides the work-groups by grouping consecutive work-item together into a wavefront. The size of a wavefront cannot be set by the developer but is hardware specific and is selected by the device vendor. During execution, each work-group is assigned to a Compute Unit. The wavefronts are executed sequentially on this compute unit with all the work-items within a wavefront executing in lockstep on the compute unit. According to the AMD OpenCL User Guide [2], the best performance is attained when the work-group size is an integer multiple of the wavefront size.

4.2.1 Address Space and Memory Hierarchy

Knowledge about the OpenCL memory hierarchy plays an important role in designing an application with good performance. In Figure 4.3, we see that, at the device-level, each device has its own global and constant memory. This memory is accessible by the host and also by all the processing elements of the device. Constant memory, is a special type of global memory (usually smaller) that is used to store read-only data. Depending on the vendor implementation, constant memory might provide faster access times than global memory. We also see that each compute unit has its own local memory and is accessible by all processing elements belonging to that compute unit. It is important to note that processing elements belonging to one compute unit cannot access local memory of another compute unit. Local memory is very fast, but is limited in size and is used to efficiently share data between work-items within the same work-group. Finally, each processing element has its own private memory (similar to registers) that cannot be accessed by other processing elements. Table 4.1 summarizes the type of access, the scope and the lifetime of the different OpenCL memory types within the OpenCL execution model.

Figure 4.5 illustrates the OpenCL memory model and how data flows between the host and the OpenCL device. Similar to the CPU memory hierarchy, there is a trade-off between

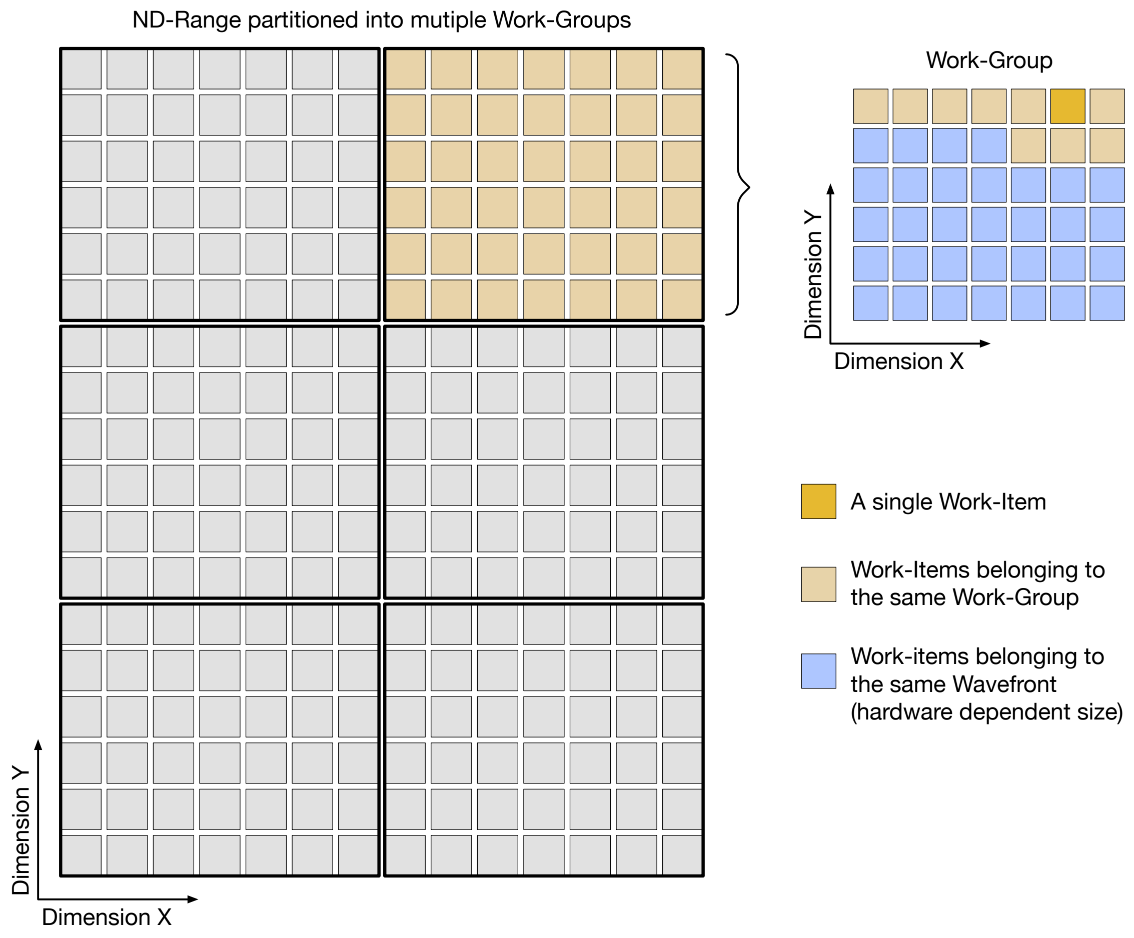

ND-Range partitioned into mutiple Work-Groups
Dimension Y
Dimension X
Work-Group
Dimension Y
Dimension X
A single Work-Item
Work-Items belonging to the same Work-Group
Work-items belonging to the same Wavefront (hardware dependent size)

Figure 4.4: The relationship between the ND-Range, work-groups, the wavefront and work-items.

Table 4.1: Scope of OpenCL memory w.r.t. the OpenCL execution model.

Memory	Access	Scope	Lifetime
private	read/write	single work-item	work-item
local	read/write	all work-items in a work-group	work-group
constant	read-only	all work-items + host	host allocation
global	read/write	all work-items + host	host allocation

the access time and storage capacity of the different memory levels. Private memory (register) is the smallest memory with the fastest access time, while global memory has the highest access time as well as the largest capacity. Economic and physical limitations play a major role in this type of memory hierarchy. To perform computation on the device, the host first needs to transfer the required data from the host memory to the OpenCL devices global memory. Once data is available in the global memory, processing elements (within a compute unit) can directly access global memory and load required data into their registers (private memory). The time required to access global memory is slow as compared to local memory. In the case of high data reuse, it is beneficial to first load a part of the global memory into faster local memory of the compute unit. Processing elements can then directly load the data from the local memory into their private memory, avoiding multiple global accesses to the same global memory and resulting in improved kernel performance. Finally, when the kernel has finished executing, the host needs to transfer data from the device's global memory back to the host. It is important to note that the host can only access the device's global memory and that if local memory was modified during the execution of the kernel, it needs to be first explicitly copied to the device's global memory.

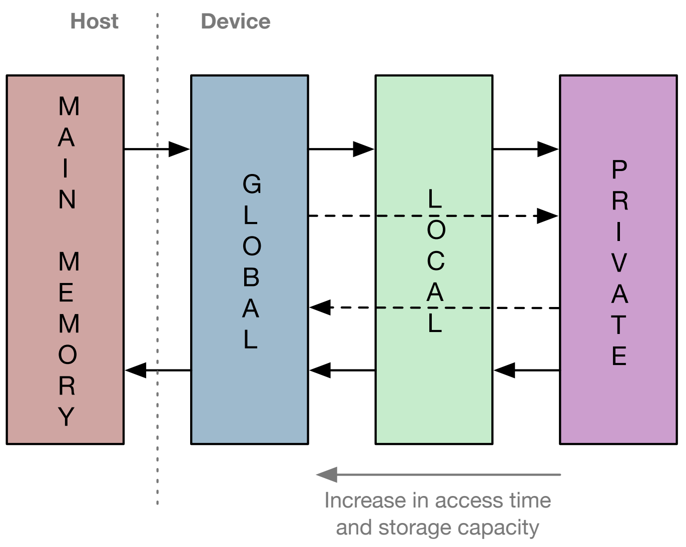


Figure 4.5: OpenCL memory model showing the data flow between the host (CPU) and the OpenCL device [2].

4.2.2 Components of an OpenCL Application

At its basis, an OpenCL application consists of the OpenCL kernel(s) and the OpenCL host code. The host is typically a CPU which controls the connected OpenCL devices. The OpenCL kernel is a parallelized function that is written using OpenCL C and executes in parallel on an OpenCL device. Kernel functions are marked with `__kernel` to identify them and also allows them to be called from the host code. A kernel on its own only represents the raw computation on a compute device. A kernel can execute on a compute unit across multiple processing elements in parallel (in lockstep). However, corresponding

OpenCL host code is required to setup the OpenCL device (create the OpenCL device context), compile the kernel (build the OpenCL program), create buffers on the device and transfer data to them, call the kernel and finally transfer data back to the host. This is a time consuming and error-prone process. In our approach, we automatically generate OpenCL C kernels and integrate them into the application by generating the corresponding OpenCL host code.

4.3 Chapter Conclusion

In this chapter, we looked at the LLVM compiler infrastructure and at its different front-ends, passes and back-ends. These different LLVM phases make it easier for a compiler developer to design and add new compiler components. The LLVM compiler infrastructure also provides an execution environment which is known as the *Execution Engine* within which an application can be executed. It also supports the JIT execution model, which is used by the RTCS to modify an already running application. LLVM makes use of LLVM IR which is a machine independent representation of a high-level program and is used as an input to the RTCS.

We also introduced the OpenCL framework that is used by the RTCS to target diverse accelerator architectures. The OpenCL standard abstracts away the architectural differences of different accelerators allowing us to target diverse accelerator architectures with the same OpenCL code. We introduced the OpenCL platform, memory and execution models and saw how an OpenCL device was made up of multiple compute units with each compute unit containing multiple processing elements. The address space and OpenCL memory hierarchy were also described. Finally, we looked at the additional steps required to integrate an OpenCL kernel into an application.

In the next chapter, we take an in-depth look into the Runtime and Just-in-Time Compilation System and how it is able to automatically parallelize sequential applications by generating and integrating OpenCL kernel and host code into an application.

CHAPTER 5

Runtime and Just-in-Time Compilation System

In Chapter 3, we saw that the run-time and self-adaptive Orchestrator forms an integral part of the *saveHSA*. The Orchestrator can manage heterogeneous accelerators for diverse applications with dynamic workloads taking into account application goals and system constraints while at the same time balancing the accelerator resources to improve performance, utilization, dependability and overall energy utilization. We also saw that in order to benefit from the *saveHSA*, application developers need to supply it with different accelerator specific implementations. Heterogeneous systems comprise of different accelerators with diverse architectures that require different programming techniques, tools and compilers. Making it difficult for application developers to port code to different heterogeneous accelerators.

In Sections 2.1 and 4.2 we saw that it is possible to target diverse architectures by using OpenCL. The application developer, however needs to first parallelize that application by writing OpenCL kernel code for the application hotspots and then write OpenCL host code that transfers data and launches the kernel on the accelerator. This manual translation of the application to use OpenCL is a time consuming and error-prone process. In this chapter, we present a Runtime and Just-in-Time Compilation System (RTCS) that automatically and transparently transforms suitable data-parallel loops into independent OpenCL-typical work-items (kernels), generates corresponding OpenCL host code and then integrates the accelerated code into the application so that it can be used by the Orchestrator to take advantage of the *saveHSA*.

To better understand how the RTCS enables the Orchestrator to target missing accelerator implementations, we look at Figure 3.3, where a SAVE-Enabled application (SEA) is executed with the help of the Orchestrator. We see that even though the platform supports multiple accelerators, the Orchestrator is only able to switch between the two implementations (here: CPU and GPGPU) that were provided by the application developer. The RTCS enables the Orchestrator to target more accelerators by transparently generating and integrating missing accelerator implementations into the application, on demand and in a JIT fashion while the application is running. This allows the Orchestrator to select from a more diverse pool of accelerators allowing for better system utilization, performance and self-adaptiveness.

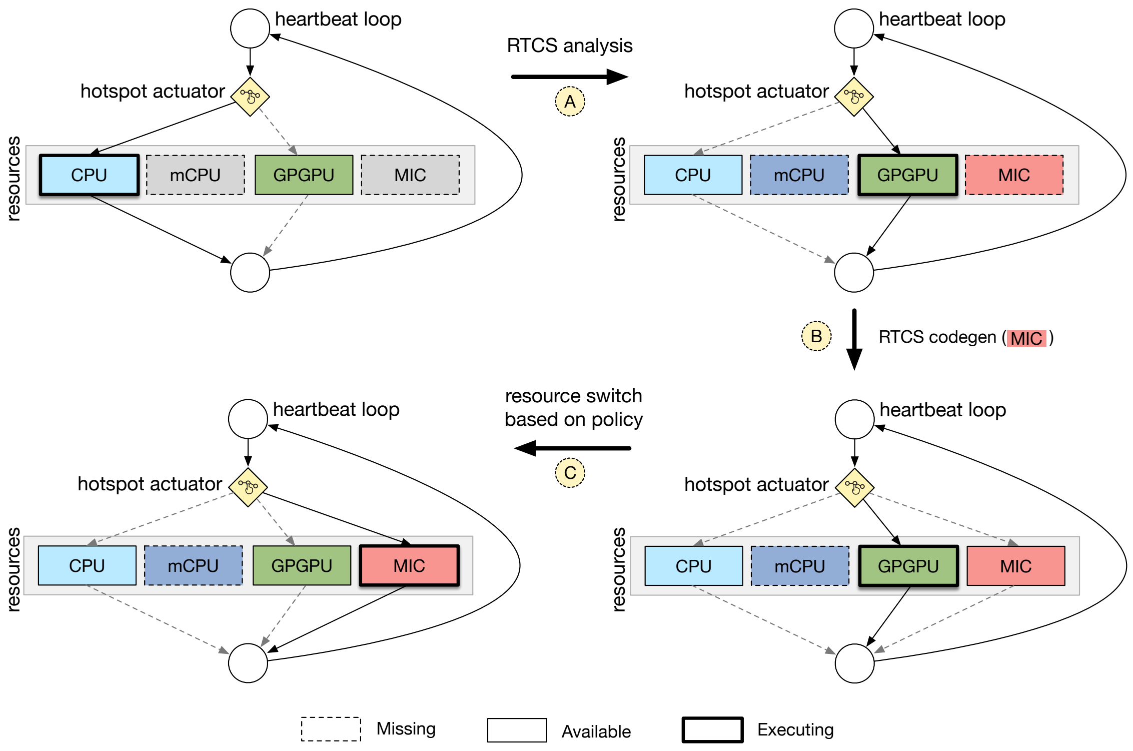


Figure 5.1: Heartbeat loop of a SAVE-Enabled application with RTCS support.

Figure 5.1 depicts the execution of the same SEA from Figure 3.3 but with RTCS support available. The RTCS analyses the application and detects the available and missing accelerator implementations (A) and offers additional code generation opportunities to the Orchestrator (mCPU and MIC). Depending on its policies, the Orchestrator instructs the RTCS to generate code for a specific platform (here: MIC). The RTCS transparently generates and integrates the requested implementation into the application (B) and registers it with the Orchestrator. The Orchestrator is now able to switch to the new accelerator implementation (here: MIC) in future loop iterations (C).

In this example, the accelerated code for the GPGPU was already available. However, developers do not need to specify any accelerator implementation at all. The RTCS is capable of generating accelerator code for different heterogeneous accelerators if the sequential CPU implementation is available. This enables applications that have only been written for CPUs, to benefit from execution on heterogeneous resources. Not only does this drastically reduce the development effort required to port applications to diverse heterogeneous resources, but also does not require the application programmer to have any accelerator specific knowledge.

5.1 Overview of the RTCS

Generating code for different types of accelerators requires different compilers and tools. However, installing and maintaining the entire compilation tool-flow for all accelerator types on each heterogeneous node is cumbersome and time consuming. The compilation times and resources (memory or CPU utilization) required to generate accelerated code vary drastically and depend on the compilers used, and in some cases cannot be determined a priori. Additionally, some compilers and tools require paid licenses which are either limited to specific machines or have restrictions on the number of concurrent users. Hence, the RTCS is separated into two main components, the RTCS Client and the RTCS Server. The client runs on the host node, i.e. the machine/node that has the accelerators attached to it. The RTCS Client provides an environment to execute, observe and analyze SAVE-Enabled applications that are present in the form of LLVM IR[1]. It is responsible for application analysis, communication with the orchestrator, integrating the generated accelerated code into the application and registering the accelerated code with the Orchestrator. While the server is responsible for parallelizing the application and generating accelerated code for different accelerators.

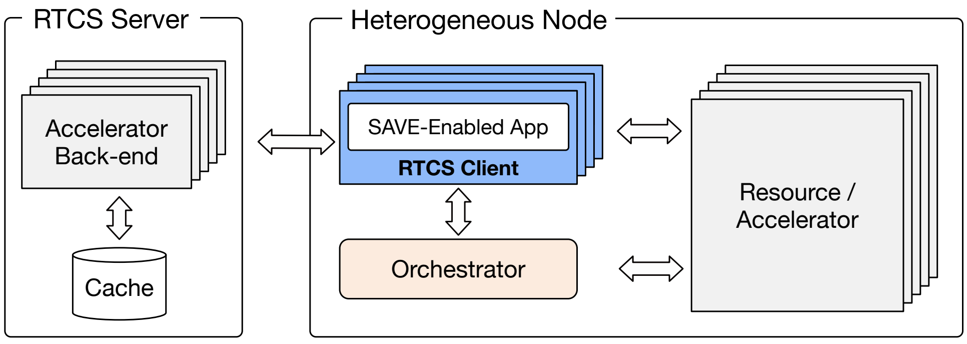


Figure 5.2: High level outline of the RTCS Client-Server architecture.

Figure 5.2 outlines the RTCS Client-Server architecture at a high level. Each heterogeneous node consists of multiple accelerators which can be targeted by the RTCS. Each SEA is executed in a separate instance of the RTCS Client. The Orchestrator runs as a separate process on the system and has a global view of all the applications running and their goals. It is also able to monitor the overall status of the system and attached accelerators. The RTCS Server is able to generate accelerated code for different accelerators by using different accelerator specific back-ends. Additionally, the RTCS Server also employs a cache to reduce subsequent code generation overheads. The RTCS Server typically serves multiple clients, and as the number of clients grow, the server might run out of resources to handle all the requests. This however is a known problem in the web-services domain and can be addressed by replicating the number of servers and using a load-balancer [16].

[1]Allows the RTCS to target all applications that can be compiled into, or are distributed in the LLVM intermediate representation

In the next section, we look at the end-to-end RTCS toolflow and how the different components of the RTCS and the Orchestrator interact with one another.

5.2 RTCS Toolflow

The RTCS is a user-transparent end-to-end approach capable of transparently accelerating sequential applications by automatically transforming suitable data-parallel loops into independent OpenCL-typical work-items (OpenCL kernels) as well as generating the corresponding OpenCL host code. This generated code is integrated into the application by exploiting the JIT functionality of the LLVM execution engine. The LLVM infrastructure is used to analyze and modify the application, while the LLVM Execution Engine is used to execute the application. The Execution Engine supports the recompilation of only parts of an application (functions) at runtime, which allows the application to be dynamically modified while it is still executing. This plays an important role in our client-side approach, allowing us to build an environment that is capable of analyzing and updating the application while it is executing.

Figure 5.3 shows the architecture of the RTCS and the order of the different steps of the toolflow (labeled with ◯). The various components of the approach are grouped into the RTCS Client and the RTCS Server with the Orchestrator being depicted as an external component. It is important to note that there are two communication channels. The green lines represent the communication between the currently running application and the Orchestrator, while the blue lines represent the communication between different RTCS components as well as the Orchestrator. In this section, we look at the general overview of the toolflow and how the different components interact with one another.

The RTCS Client is invoked from the console and acts as an execution container for a SAVE-Enabled application, with each SEA having its own client instance. The client accepts the application binary of the SEA as platform-independent LLVM bitcode ① along with its arguments. The bitcode is parsed into an in-memory representation ② and canonicalized ③ so that further custom LLVM analysis and transformation passes can be applied. The LLVM IR is then analyzed to detect the different accelerator implementations provided by the application along with the Orchestrator interfaces used by the application ④. It should be noted that the SEA needs to provide a single-threaded CPU implementation for the RTCS to be able to generate accelerated code. All the analysis information is stored in a central Hotspot Data Structure (HDS) that can be accessed by all the RTCS components. The application hotspots are then analyzed ⑤ to identify data-parallel loops and the HDS is updated. At this point, the client knows which accelerator implementations have been provided by the application developer and which ones are missing. To help with the integration of the generated accelerated code at runtime, the client inserts accelerator hooks into the application for all the missing accelerator implementations and also adds OpenCL support to the application ⑥.

Once the application has been successfully modified, the client commences executing the application via the LLVM Execution Engine ⑦. When the application starts executing, it communicates its goals and available implementations to the Orchestrator via the ⑧

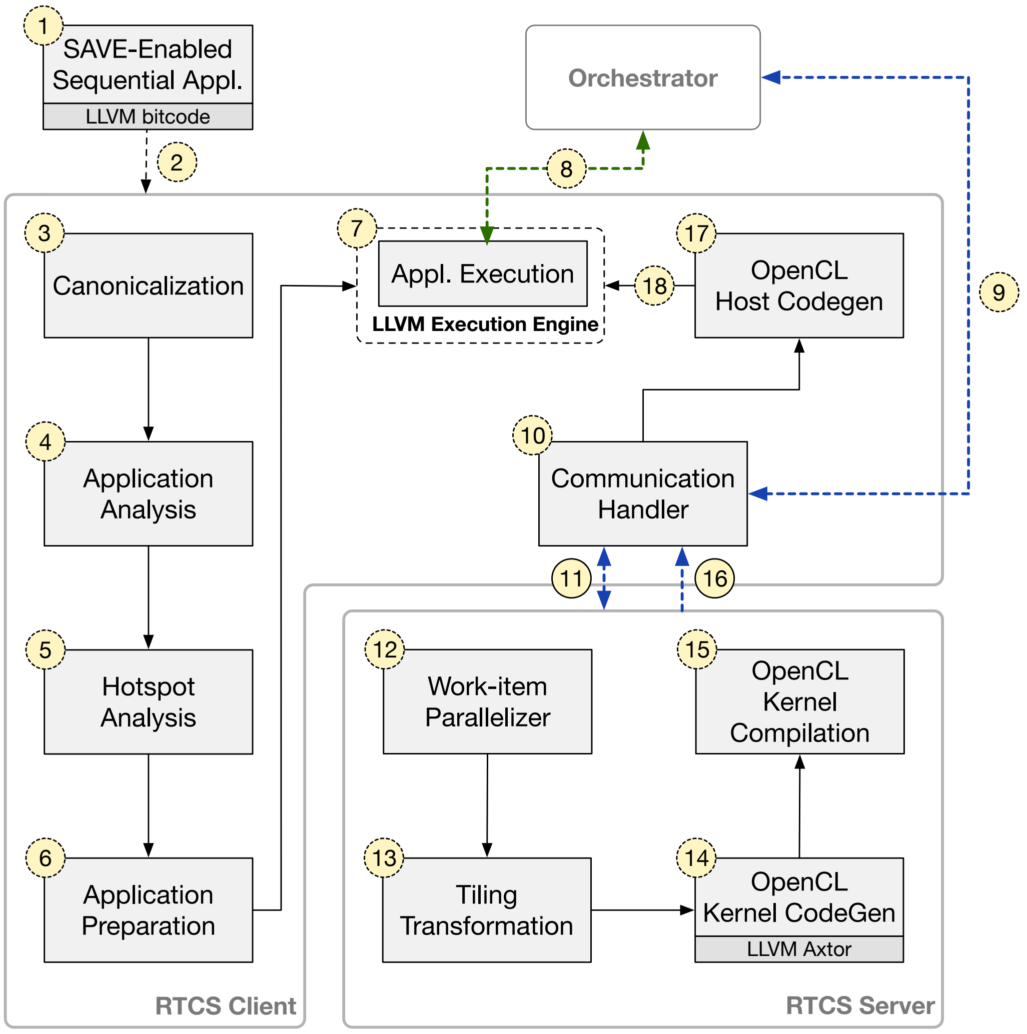


Figure 5.3: Overview of the RTCS compiler architecture.

communication channel. The application also uses this channel to report its performance (via heartbeats) to the Orchestrator. Additionally, the Orchestrator also instructs the application to switch to another accelerator using the same communication channel. The Orchestrator communicates with the client using the (9) communication channel and is able to instruct the client to generate code for a specific accelerator. The communication handler module (10) on the client handles all requests from the Orchestrator as well as the RTCS Server. On receiving a code generation request from the Orchestrator, the client forwards the code generation request to the RTCS Server (11).

The RTCS Server runs as a separate application/process and is responsible for parallelizing the hotspot and generating accelerator specific code. On start-up it begins listening for clients that want to connect to it and on a successful connection, it spawns a new thread that handles all further requests from that client. On receiving a code generation request for an accelerator ⑪, the RTCS Server first checks its cache to see if the code has been previously generated. By using such a cached approach, the code generation overheads for subsequent requests can be drastically reduced by skipping ⑫ - ⑮. If the request cannot be satisfied by the cache, the required accelerator code is JIT generated. Depending on the target accelerator, the Work-item Parallelizer ⑫ transforms suitable data-parallel loops of the hotspot into independent OpenCL-typical work-items that are executed in parallel. The Tiling Engine ⑬ then analyses the hotspot for data reuse patterns and attempts to tile the hotspot for more efficient local memory reuse. The OpenCL Kernel CodeGen Module ⑭ uses the LLVM OpenCL back-end Axtor to generate the OpenCL C kernel code. This OpenCL C kernel code is then compiled into an accelerator specific binary ⑮. This newly generated accelerated code is added to the cache and transferred to the RTCS Client ⑯.

On receiving the accelerated code from the server, the corresponding OpenCL host code is generated ⑰. OpenCL API calls that create buffers on the device, transfer the data to and from the device, map the kernel arguments and invoke the kernel, are inserted into the appropriate accelerator hook (previously created in ⑥). This modified accelerator hook (function) is JIT compiled and integrated ⑱ into the running application with the help of the LLVM Execution Engine. In the next iteration of the heartbeat loop, the new accelerator implementation is registered with the Orchestrator, which allows the Orchestrator to offload computation to a previously unavailable accelerator.

Interaction between the Orchestrator and the RTCS: To help understand and visualize this toolflow, we make use of the example previously illustrated in Figure 5.1. We saw that the SEA natively only supported the CPU and GPGPU implementations and how the RTCS enabled the Orchestrator to also target the MIC by transparently generating and integrating accelerator code for the MIC. Figure 5.4 shows the detailed interaction between the Orchestrator and the different components of the RTCS Client and the RTCS Server.

The Orchestrator is an operating system process that is always running in the background. The RTCS Server runs as a separate process, is initialized and waits for incoming connections from the client(s). The RTCS Client acts as an execution environment for SAVE-Enabled applications and all SEAs are launched on the heterogeneous node via the client. The client loads the LLVM IR ② and canonicalizes ③ it. It then analyses the application, analyses the hotspots and prepares the application by adding accelerator hooks and OpenCL support to the application (④ - ⑥). It also registers the application with the Orchestrator ⑨ and establishes a connection to the server ⑪. The server uses a separate thread for each client and spawns a new thread to handle the new connection. The client then compiles the application and starts its execution using the LLVM execution engine ⑦. The application initializes and then registers the available implementations (here: CPU and GPGPU) with the Orchestrator ⑧. The Orchestrator can now select on which resource the application should execute on.

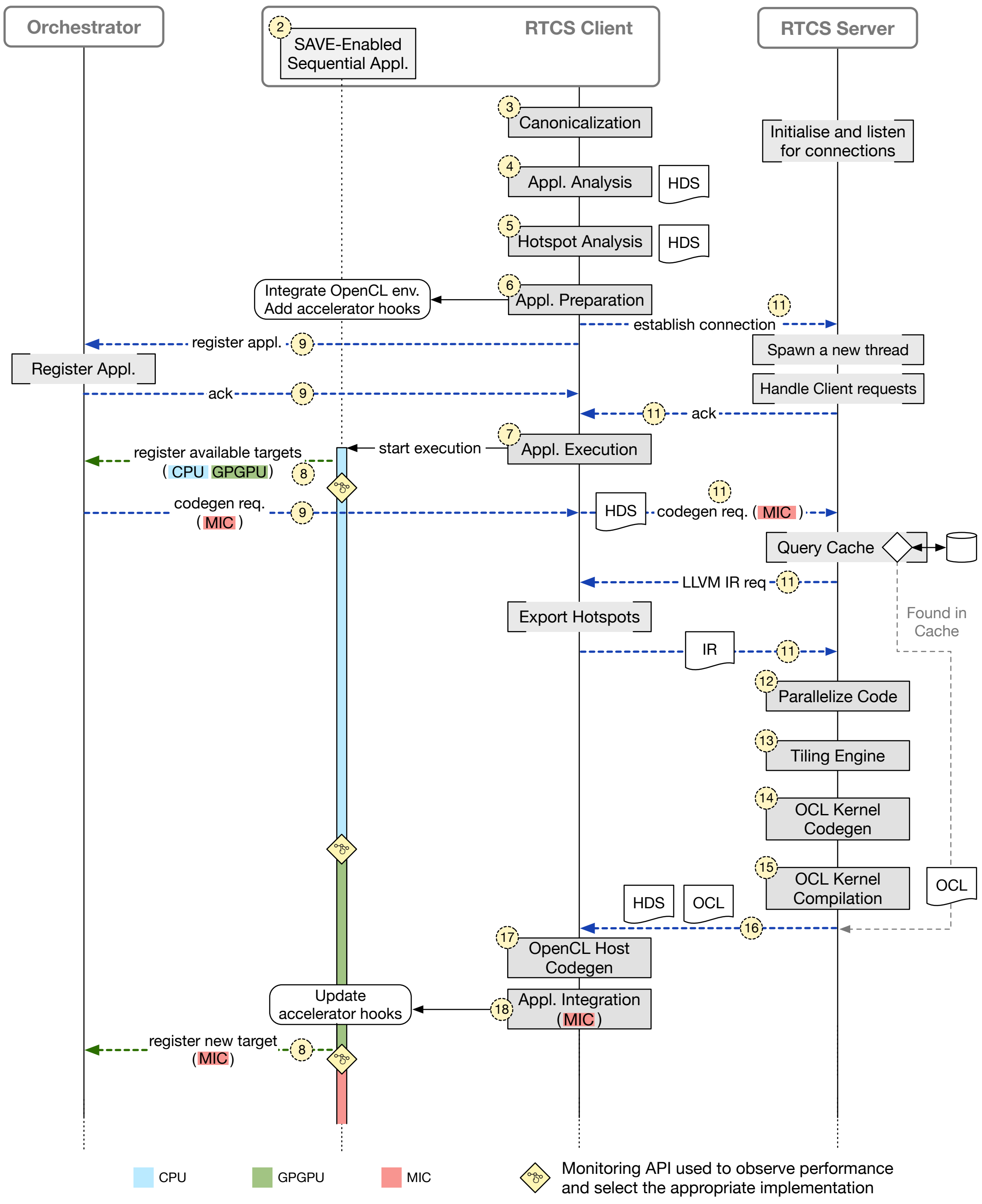


Figure 5.4: The interaction of the Orchestrator, the RTCS and the SAVE-Enabled application. For the sake of simplicity, the interaction between the SAVE-Enabled application and the Orchestrator during ◇ are not shown.

At any time during the application's execution, the Orchestrator can instruct the RTCS Client to generate code for a specific accelerator ⑨ (here: MIC). The RTCS in turn requests the server to generate code for the specific accelerator ⑪. If the server cannot find the requested code in its cache, it asks the client for the corresponding sequential LLVM IR code of the hotspots. The client exports the required code into a new LLVM module and sends it to the server. The server then parallelizes the sequential code, performs data optimizations, generates the OpenCL C kernel code and creates an accelerator specific binary (⑫ - ⑮). The accelerated code along with the modified HDS is transferred to the client ⑯, which generates the corresponding OpenCL host code ⑰ and integrates the new implementation into the application using the JIT compilation feature of the execution engine ⑱. The application then registers the new implementation with the Orchestrator, allowing the Orchestrator to switch to the new accelerator implementation (here: MIC) in future loop iterations.

The RTCS has been designed to be modular, allowing different components to be replaced or updated at a later date with state-of-the-art versions. In the following sections, we take a detailed look at each of the components and see how they contribute to the Runtime and Just-in-Time Compilation System. Since most of the components rely on the communication infrastructure in one way or another, we first take a look at the communication module.

5.3 Communication

The Orchestrator and the SAVE-Enabled application natively communicate with one another using the Orchestrator API which is based on the shared memory communication model ⑧. The application is able to communicate its performance to the Orchestrator via heartbeats, and depending on its policies, the Orchestrator instructs the application to offload computation to a specific accelerator.

The RTCS Client and the Orchestrator also need a communication channel so that the Orchestrator can instruct the RTCS Client to generate accelerator specific code ⑨. Additionally, the RTCS Client and Server also need to communicate with one another (⑪ and ⑯) in order to generate accelerator specific code. The RTCS Server requires the client to send it the Hotspot Data Structure as well as the corresponding LLVM IR of the sequential hotspot that is to be accelerated. The Orchestrator and the RTCS Client always execute on the same physical machine, however the RTCS Server can run on any node in the network. Hence a *message passing* communication model is used to handle communication between the Orchestrator, the RTCS Client and the RTCS Server.

Our communication infrastructure utilizes TCP/IP sockets to communicate and exchange data between the Orchestrator, the RTCS Client and the RTCS Server. TCP/IP is a stream-based transport protocol that guarantees an ordered data stream but does not preserve message boundaries. This means that if the sender sends two separate messages "Hello" and "World", the receiver might get "HelloWorld" as one message or might receive it in multiple messages. It might first get "He" and then "lloW" followed by "orld".

For TCP the `send()` and `recv()` API calls do not have a 1-to-1 correlation between them. Depending on the length of the sent message and how the message is distributed into packets, the receiver might need to issue multiple `recv()` calls until it has the entire message. To help simplify this, we developed a library that has a custom message format and parser that transparently handles message passing. Figure 5.5 shows the custom message format used to send and receive messages. The message is comprised of a fixed size header and a variable payload. The header consists of the message type and the payload size. Internally, the message is sent in two parts. The header is first sent to the receiver so that it knows the payload size in advance. The library allocates enough memory to receive the payload from the client. Since the payload message size is known, the library issues multiple recv() calls until the complete payload message is received. The message payload might contain the HDS, a codegen request, the sequential LLVM IR code, the accelerated code or other control and acknowledgement messages. The `MessageType` in the header is used to distinguish these different types of messages.

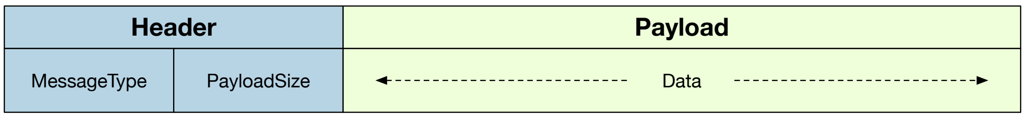


Figure 5.5: Structure of the custom message format.

To transmit the required data from the sender to the receiver, the data structure/object needs to be first serialized by the sender, transmitted to the receiver and is then de-serialized by the receiver. To simplify the serialization/de-serialization of objects that need to be transmitted, *Protocol Buffers* (ProtoBuf[2]) are used. Protobuf is an open-source project developed by Google and is a language-neutral, platform-neutral extensible mechanism for serializing structured data. To use it, the data-structures that need to be serialized are defined in a `.proto` file. The protocol buffer compiler uses this file to generate corresponding custom serialization APIs/functions that allow one to easily serialize/de-serialize complex data structures. The custom messaging library along with protocol buffers form the basis of the communication infrastructure that is used to communicate between the Orchestrator, the RTCS Client and the RTCS Server.

5.4 Hotspot Data Structure

Another vital component of the RTCS is the Hotspot Data Structure (HDS), with most of the RTCS components relying on it in one way or another. The HDS stores all the information obtained during the analysis phase which is later used by different RTCS components. The HDS also stores the housekeeping data required by the RTCS to generate accelerated code and integrate it into the application at runtime. For the sake of simplicity and to help understand the HDS, a schematic representation of the HDS is depicted in Figure 5.6.

[2]https://developers.google.com/protocol-buffers/

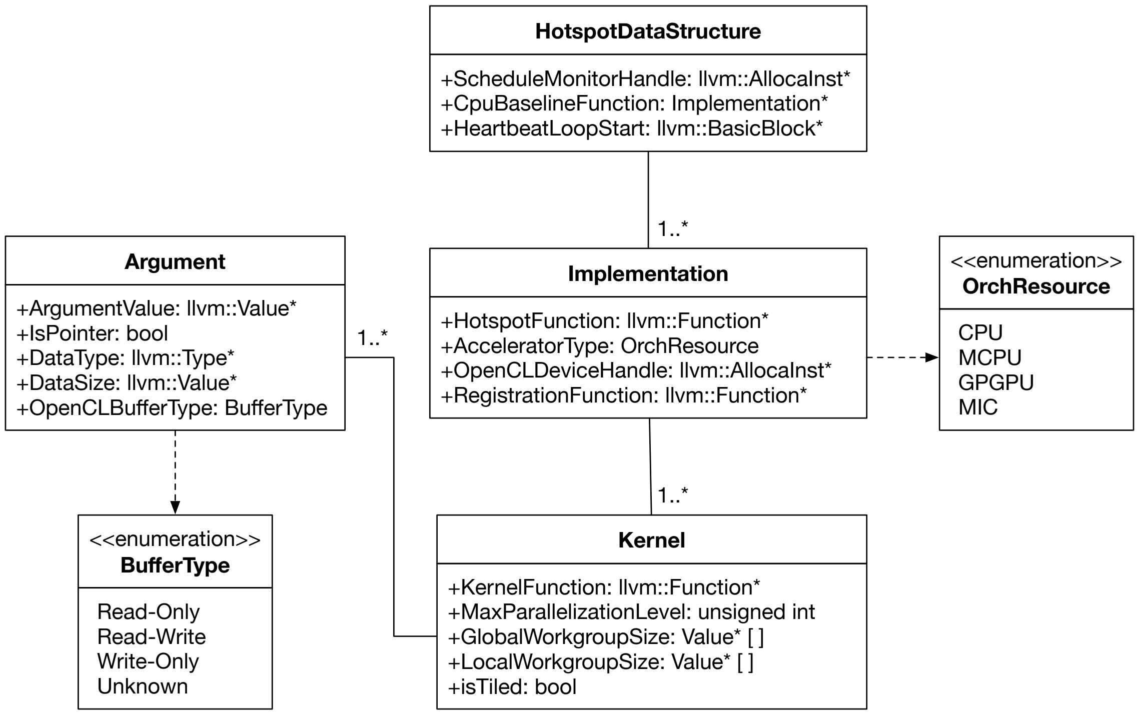

HotspotDataStructure
+ScheduleMonitorHandle: llvm::AllocaInst*
+CpuBaselineFunction: Implementation*
+HeartbeatLoopStart: llvm::BasicBlock*
1..*
Argument
+ArgumentValue: llvm::Value*
+IsPointer: bool
+DataType: llvm::Type*
+DataSize: llvm::Value*
+OpenCLBufferType: BufferType
1..*
Implementation
+HotspotFunction: llvm::Function*
+AcceleratorType: OrchResource
+OpenCLDeviceHandle: llvm::AllocaInst*
+RegistrationFunction: llvm::Function*
<<enumeration>>
OrchResource
CPU
MCPU
GPGPU
MIC
1..*
<<enumeration>>
BufferType
Read-Only
Read-Write
Write-Only
Unknown
Kernel
+KernelFunction: llvm::Function*
+MaxParallelizationLevel: unsigned int
+GlobalWorkgroupSize: Value* [ ]
+LocalWorkgroupSize: Value* [ ]
+isTiled: bool

Figure 5.6: A schematic representation of the Hotspot Data Structure.

The HDS is initially populated by the *Application Analysis* module which identifies the orchestrator constructs, the entry to the heartbeat loop and the different hotspot implementations that are provided by the application developer (see Section 5.6). The *Hotspot Analysis* module updates the HDS by adding information about the different kernels, their maximum parallelization level, the arguments used, the data-type and size to name a few (see Section 5.7). The HDS not only stores the result of the analyses performed, but is also used by subsequent phases of the RTCS to store hotspot as well as kernel specific data.

5.5 Canonicalization

In Section 4.1 we saw that different front-ends can be used to generate LLVM IR from high level languages. The LLVM IR generated by such front-ends is not standardized and vary from one front-end to another. Also, developers have different coding styles. Some might choose to use *while* loops while others might choose to use *for* loops instead. These differences in high level code lead to different structures in the resulting LLVM IR. Additionally x86 binaries can also be disassembled into LLVM IR [3] resulting in an intermediate representation that might have an ad hoc structure with more than one possible representation in LLVM IR.

Before analyzing the application, the RTCS parses the LLVM IR into an in-memory

LLVM module ② and then uses built-in LLVM passes to canonicalize ③ the given LLVM IR. These transformations canonicalize the LLVM IR structure after which further custom analysis and transformation passes can be applied. To help with hotspot analysis of the application, the RTCS also normalizes the loop structure using the `LoopSimplify` LLVM pass which transforms natural loops into a simpler form. It guarantees that there is a single entry point to the loop (pre-header) that is always executed before entering the loop. The `LoopSimplify` pass transforms the loop structure, which makes subsequent analyses and transformations simpler and more effective [59]. To further simplify loop analysis, the `IndVar` LLVM pass is applied to transform the induction variables (and computations derived from them) into simpler forms resulting in a normalized loop structure that is used for hotspot analysis. Finally, built-in LLVM passes are used to perform lightweight but aggressive, high-quality code optimizations. Table 5.1 represents the passes used and gives a brief description of what they do.

Table 5.1: Built-in LLVM canonicalization and optimization passes used by the RTCS.

LLVM Pass	Description
PromoteMemoryToRegister	Limits the use of memory which increases the effectiveness of subsequent LLVM passes
LoopSimplify	Canonicalizes natural loops which reduces the burden of writing loop specific passes
LCSSA	Keeps the effect of subsequent loop optimizations local which limits the overhead for maintaining the SSA form
IndVar	Normalize the induction variables and highlights the canonical induction variable
LICM	Moves loop invariant code outside the loop
ConstantPropagation	Merges duplicate global constants
InstructionCombining	Combines redundant instructions together
DeadCodeElimination	Eliminates dead code

5.6 Application Analysis

The structure of a typical SAVE-Enabled application was introduced in Figure 3.1. At the very least, a SEA is required to implement the Orchestrator API and be able to report its execution progress in the form of heartbeats. It might also provide different accelerator-specific implementations for the Orchestrator to offload to. Additionally, it might also set application-level goals and/or constraints. The Heartbeat loop is key to SEAs, where in each iteration, the Orchestrator has an opportunity to modify the behavior of the system by switching between different available accelerator implementations. To analyze the SEA and to extract relevant information from the application, we have written a custom `OrchDetect` LLVM analysis pass ④ that the RTCS uses to identify the following application constructs:

- *Scheduler Monitor* instance.
- CPU implementation of the hotspot: mandatory and used to generate accelerated code.
- Call parameters: arguments used during the invocation of the hotspot.
- Other registered implementations: optional accelerator implementations provided by the application programmer.
- Heartbeat loop: control loop where the Orchestrator can switch between resources and monitor application performance.

The Orchestrator's *Scheduler Monitor* class is responsible for managing all accelerator registrations. This class exposes the `registerImplementation()` API, which is the primary Orchestrator API via which the application registers different accelerator specific hotspot implementations with the Orchestrator. Different accelerator implementations are provided by the application developer in the form of lambda expressions [52] which are then registered with the Orchestrator via the `registerImplementation()` API call. Listing 5.1 shows an example of how such a lambda function looks like and how it is registered with the Orchestrator. The lambda expression on line 4 calls the sequential CPU implementation with a fixed set of parameters. This lambda expression is then registered with the Orchestrator via the `registerImplementation()` API call in line 9 and is set as a CPU implementation type.

```
1  int main(int argc, char* argv[]){
2      ...
3      // lambda expression for the sequential CPU implementation
4      ImplementationType cpuImpl = [&] () -> void {
5          heavyFunc (in_img0, in_img1, out_img, rows, cols);
6      };
7      ...
8      //Register the implementation with the Orchestrator
9      schedulerMonitor->registerImplementation(cpuImpl,CPUType);
10     ...
11 }
```

Listing 5.1: An example of an implementation represented as a lambda expression and its registration with the Orchestrator.

The `OrchDetect` pass analyses the application to detect the `registerImplementation()` API call and in turn, the handle to the instance of the *Scheduler Monitor* class (`schedulerMonitor` in Listing 5.1, line 9). This `schedulerMonitor` handle is saved in the HDS and is later used by the RTCS to call the `registerImplementation()` API and add missing accelerator implementations to the application (see Section 5.13).

The `OrchDetect` pass also analyses the application for the usage of the `registerImplementation()` API call to determine the different types of hotspot implementations that the application registers with the Orchestrator. This information is stored as a part of the HDS. The implementation registered as the *CPUType* represents the sequential CPU implementation and is saved as the `CPUBaselineFunction` (`heavyFunc` in Listing 5.1). This `CPUBaselineFunction` is used later on by the RTCS to generate accelerated code.

After the new accelerated code (function) has been generated by the RTCS, it needs to be called with the appropriate arguments. The lambda-expression registered as the *CPUType* is analyzed and the corresponding calling arguments are saved in the HDS. The `OrchDetect` pass also analyses the application to identify the heartbeat loop and saves the entry to the heartbeat loop into the HDS so that it can be later used to register new accelerator implementations with the Orchestrator (Section 5.8).

The `OrchDetect` pass detects the Orchestrator APIs and the sequential `CPUBaselineFunction`. However, if they are not detected, the application is not considered to be a SEA and all further steps are skipped. In this scenario, the RTCS executes the application as a plain old sequential application and the application does not benefit from our approach. However, if the sequential `CPUBaselineFunction` is found, it is used as the basis for generating the missing accelerator implementations.

5.7 Hotspot Analysis

One of the challenges of automatic parallelization is to identify the computationally intensive parts of the application and partition the application accordingly. In [Vaz16] we demonstrate how applications could be partitioned by using a heuristic based code analysis and profiling method to identify computationally intensive parts of an application. A good indicator for potential hotspots is loops as they can have very long run-times and show a monotonic execution behavior. In [Ken14] we demonstrated how an application could be partitioned by analyzing basic blocks within loops and extracting them to a separate function. We also looked at a polyhedral model (Polly) based partitioning approach in [Rie19] where Static Control Parts (SCoPs) were used to automatically identify hotspots in an application. However, in the case of SAVE-Enabled applications, the Orchestrator can only switch between different accelerated versions of the `CPUBaselineFunction` allowing us to treat the entire `CPUBaselineFunction` as a hotspot. However, such a hotspot might internally consist of one or more function calls (referred to as kernels). Listing 5.2 shows how such a hotspot function for a motion detection application might look like. In this case, the hotspot has two kernels forming a chain with data produced by one kernel and consumed by the subsequent kernel(s). The hotspot analysis phase analyses the `CPUBaselineFunction` (hotspot) as well as the comprised kernel(s), and updates the HDS to hold the analysis information for the hotspot as a whole, as well as the individual kernels.

Custom LLVM analysis passes were developed to analyze the properties of the hotspot kernels ⑤ and are explained in detail in the following sections.

```c
void heavyFunc (char* in_img0, char* in_img1, char* out_img, int rows,
    int cols){
  char *imgA = (char *) malloc (rows * cols * sizeof (char));
  char *imgB = (char *) malloc (rows * cols * sizeof (char));
  char *imgC = (char *) malloc (rows * cols * sizeof (char));

  kernel_gauss (in_img0, imgA, rows, cols);
  kernel_rgb2grey (imgA, imgB, rows, cols);

  // Apply same filters on image 1.
  kernel_gauss (in_img1, imgA, rows, cols);
  kernel_rgb2grey (imgA, imgC, rows, cols);

  // Compute motion difference.
  kernel_motion (imgB, imgC, imgA, rows, cols);

  // Overlay results in input image.
  kernel_greyedge2rgb(in_img0, imgA, out_img, rows, cols);

  free(imgA);
  free(imgB);
  free(imgC);
}
```

Listing 5.2: An example of a hotspot (`CPUBaselineFunction`) that contains multiple kernels from a SAVE-Enabled application that detects the motion between two images.

5.7.1 External Function Filter

OpenCL accelerators (devices) have limitations regarding what type of code can be executed on them and they do not support system calls, shared library calls and calls to functions on the host. The `ExternalFunctionFilter` pass analyses the hotspot kernels for system calls and library calls. Functions that have `external_weak/external` as their LLVM linkage type are deemed to be external functions. The `ExternalFunctionFilter` pass examines all the call instructions within a kernel, determines the called function and checks its linkage type. If a call to an external function is found, the entire hotspot is marked as not viable for acceleration and all further analysis are skipped. OpenCL however, supports calls to `math` functions and the `ExternalFunctionFilter` pass maintains a white-list containing all the math functions which it treats as exceptions.

5.7.2 Dependence Analysis

The RTCS accelerates sequential code by automatically parallelizing natural loops inside hotspot kernels. For a loop to be parallelized, it should be able to execute each loop iteration independently and cannot have any cross-iteration data dependencies. This means that a loop needs to be free of data dependencies between each loop iteration. If this condition is met, the RTCS can guarantee that the loop is safe to execute in parallel and hence can be parallelized. Listing 5.3 shows an example of a simplified 2D 3×3 convolution. The outer two loops (lines 2-3) iterate over the 2D output space, while the inner two loops (lines 6-7) perform a 3×3 convolution for each entry.

Looking at the pseudo C code in Listing 5.3, we can see that the input `in` and output `out` are independent of each other. We also see that the outer two loops (lines 2-3) do not have inter-loop dependencies and can be parallelized. However, in the inner convolution loops (lines 6-7), the current value of `sum` depends on the previous value of `sum`, and hence the inner loops cannot be parallelized. To determine if there are any loop dependencies, our custom `DependenceAnalysis` pass first analyses all the memory accesses. It does this by looking at all the instructions and checks if the instruction reads (`load`) from or writes (`store`) to memory. It creates a memory access structure by analyzing every memory instruction and storing the loop that it belongs to, the base array pointer variable and the corresponding array access index. Using this information, the pass determines the data read/write dependencies between the different memory accesses and also the inter-loop data dependencies. The `DependenceAnalysis` pass is able to determine if the hotspot can be parallelized and calculates the maximum parallelization loop level. In our example, since the outer two loops are independent, the maximum parallelization level is 2. Additionally, the kernel is also analyzed for co-dependence (j loop iterator does not depend on i). If no independent loops are found, the entire hotspot is marked as not viable for acceleration and all further analysis are skipped.

```
 1  // Iterate over the 2D output space
 2  for (int r = 0; r < rows - 2; r++) {        // independent
 3    for (int c = 0; c < cols - 2; c++) {      // independent
 4      // Convolve over the 3X3 filter.
 5      int sum = 0;
 6      for (int i = 0; i < 3; i++) {           // dependent on sum
 7        for (int j = 0; j < 3; j++) {         // dependent on sum
 8          // ...
 9          sum += in[r+i][c+j] * MASK[i][j];
10      // ...
11      out[r][c] = sum;
```

Listing 5.3: Pseudo C code for a 3×3 convolution demonstrating loop dependencies.

To perform this dependence analysis, it is important that there are no aliases present. This means that two different data-arrays do not point to the same memory location. For example, in Listing 5.3, `in` and `out` should not point to the same memory address. The `DependenceAnalysis` pass itself does not perform any alias analysis, but to guarantee that no aliases are present, it checks if the base array pointer variables have been defined with the `__restrict` type qualifier (reflected by the `noalias` attribute in LLVM IR). If, however, the `__restrict` keyword was not used and the user knows that no aliases are present, this check can be disabled via a command line switch when launching the RTCS Client.

5.7.3 OpenCL Data Buffer Analysis and Data Transfer Optimizations

Before an OpenCL kernel can be executed on the accelerator, the appropriate data buffers need to be created on the OpenCL device. Additionally, the required data needs to be transferred to the device and after the execution is complete, data needs to be transferred back to the host. The `DataOptimization` analysis pass uses the memory access information created by the `DependenceAnalysis` pass to determine which data-arrays are used by the kernels. It then analyses the application to determine where the data-arrays were defined and determines their data-type and size.

The `DataOptimization` pass analyses the memory access of the data-arrays within the kernel to determine if a data-array is read from or written to memory. It classifies every data array as read-only, read-write or write-only. This is then used to optimize the OpenCL data buffer creation on the device along with optimized data-transfers to and from the device. In the 3×3 convolution example in Listing 5.3, we can infer that array `in` and `MASK` have read-only memory accesses (line 9). If a data-array is read-only, it is never transferred back to the host as the data was not modified.

We can also see that array `out` has a write-only memory accesses (line 11) in Listing 5.3. Since it is a write-only data-array, one would be tempted to optimize the data transfer by not transferring the data-array to the accelerator and only transferring the data back after the computation is finished. However, we cannot guarantee that all the elements of such a data-array are modified on the accelerator and hence we are not allowed to perform such a data transfer optimization. In our example, the RTCS transfers the `in`, `out` and `MASK` arrays to the device. While only the `out` array is transferred back to the host after the computation is complete on the device.

If we have multiple kernels in a hotspot, every kernel needs to transfer the required data to the device, perform the computation and then transfer the data back to the host. This results in repeated data transfers to the device if a hotspot has multiple kernels that use/re-use the same overlapping data from one another. Data needs to be transferred back when kernel calls are interleaved with other (sequential) computations in the hotspot function. However, if there are no additional computations in the hotspot (like in Listing 5.2), the data transfer between the host and the device can be optimized. Looking at the hotspot example in Listing 5.2 we can see that `kernel_rgb2grey` uses the data generated by `kernel_gauss`, similarly `kernel_motion` uses the data generated by `kernel_rgb2grey`. Instead of always transferring data to and from the host for every kernel, the `DataOptimization` analysis pass performs inter-kernel data analyses to determine what data optimizations can be applied across multiple kernels. Figure 5.7 depicts the execution of a data optimized kernel chain on an accelerator. All the required data is first transferred to the device, the OpenCL kernels are then executed on the device and finally only the modified data is transferred back to the host.

Looking at Listing 5.2 we can also see that `imgA`, `imgB` and `imgC` are only declared within the scope of the hotspot. If there are no additional computations within the hotspot, the `DataOptimization` analysis pass identifies such data-arrays whose scope are only limited to the hotspot. In these cases, data buffers are only created on the device and data is never

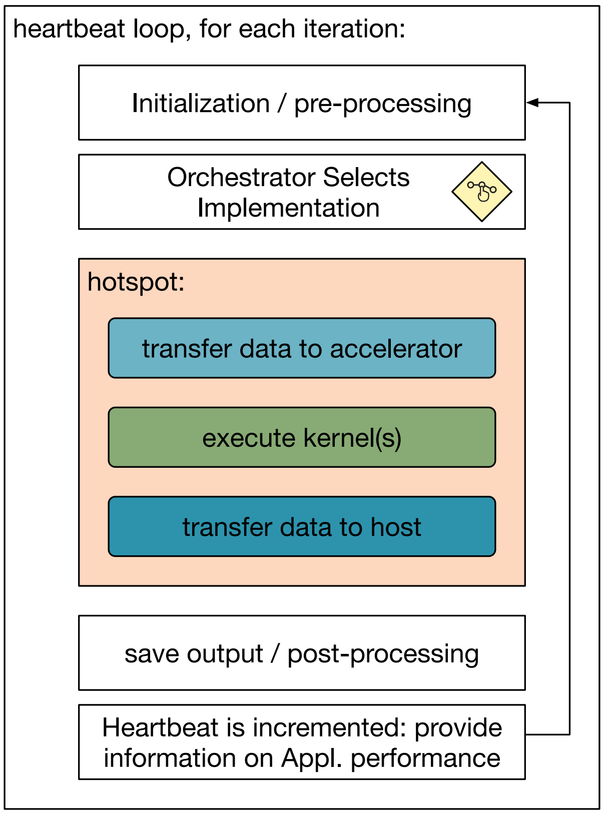

heartbeat loop, for each iteration:
Initialization / pre-processing
Orchestrator Selects Implementation
hotspot:
transfer data to accelerator
execute kernel(s)
transfer data to host
save output / post-processing
Heartbeat is incremented: provide information on Appl. performance

Figure 5.7: The structure of a data optimized kernel chain when executed on an accelerator.

transferred to the OpenCL device or transferred back to the host.

This analysis information is stored as a part of the Hotspot Data Structure and is later used to generate OpenCL host code in Section 5.13. In the next section we take a look at how the application is prepared and executed via the LLVM execution engine.

5.8 Application Preparation and Execution

The Orchestrator is a self-adaptive operating system layer that enables the *saveHSA* system to efficiently manage heterogeneous accelerators. It does this by dynamically offloading workloads to available resources by taking into account the varying workloads, application goals, system constraints and resource availability. During the lifetime of the application, the Orchestrator might request the RTCS to generate a missing accelerator implementation. The RTCS is capable of generating and integrating the accelerated code into the application (see Section 5.13). However, before the Orchestrator can take advantage of the new accelerator implementation, the application needs to first register this new implementation with the Orchestrator. In Section 5.6, we saw that the registration is performed by calling the Orchestrator's registration API with a lambda function and the accelerator type as parameters. This registration call needs to be inserted into the application and has to be called with the execution context of the application.

Adding accelerator hooks: In a SEA, the heartbeat loop is where most of the computation takes place. If code is inserted into this heartbeat loop, it is guaranteed to execute for every iteration of the heartbeat loop. Hence, whenever a new accelerator implementation is generated at runtime, it can be registered with the Orchestrator if the registration API is called from inside this loop. However, altering the control flow of the heartbeat loop at runtime is not a trivial task and hence the application is modified before it is executed. The RTCS determines the missing accelerator implementations from the analysis information previously stored in the HDS. For each missing implementation, the RTCS creates a corresponding registration function with the appropriate signature. Calls to the newly created registration functions are then added to the beginning of the heartbeat loop. The registration calls are inserted before the Orchestrator API call that selects which accelerator implementation to execute, allowing the Orchestrator to select the new implementation in the same heartbeat cycle.

Figure 5.8 shows the registration functions that were added for the missing acceleration implementations. These registration functions are called during every heartbeat iteration. However, initially these registration functions are empty and don't do anything useful when called from within the heartbeat loop. However, at application runtime, after a new accelerator implementation is generated, the RTCS updates the appropriate registration function with the corresponding lambda function and a call to the Orchestrator's registration API. This updated registration function is then JIT compiled using the LLVM Execution Engine and the registration function pointer is updated to point to the newly compiled function. When the updated registration function is now called in the next heartbeat iteration, the new accelerator implementation is registered with the Orchestrator. Multiple registrations of the same implementation during subsequent heartbeat iteration calls are avoided by registering a new implementation only when this updated registration function is called for the first time (see Section 5.13). By inserting accelerator hooks during the application preparation phase, the RTCS simplifies the process of registering new implementations with the Orchestrator at application runtime.

Adding OpenCL support: The entire RTCS toolflow relies heavily on OpenCL and hence the RTCS needs to add OpenCL support to the SEA. We have implemented an OpenCL wrapper library that adds common OpenCL API headers and creates OpenCL device handles and command queues for all the OpenCL devices supported by our evaluation platform (described in Section 6.1). This step is only performed once during the application lifetime, and the device handles and command queues are reused by the RTCS. The SEA is executed by the LLVM Execution Engine inside the address space of the RTCS Client. The RTCS Client is linked against the appropriate OpenCL libraries which allows the inserted OpenCL API calls to be dynamically resolved.

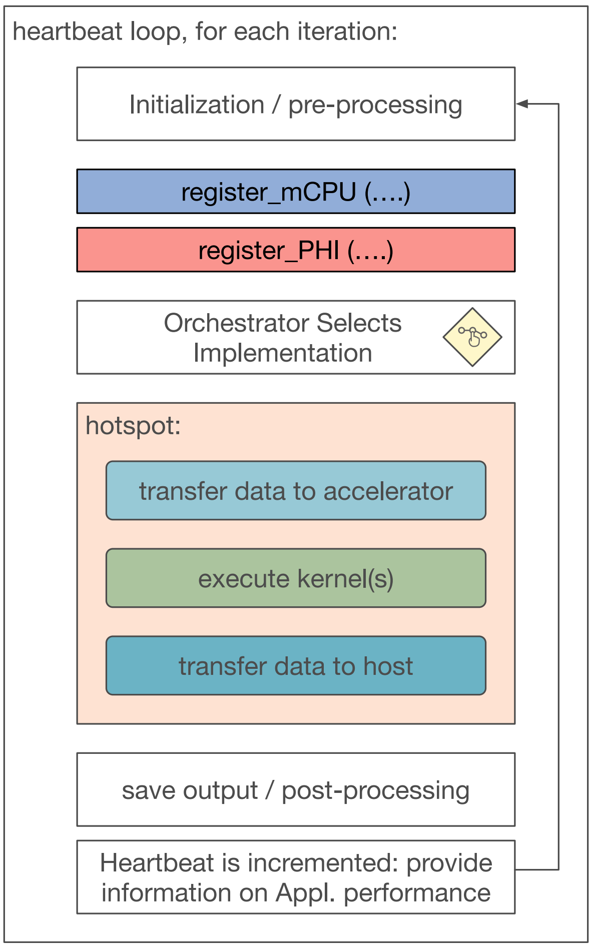

heartbeat loop, for each iteration:
Initialization / pre-processing
register_mCPU (....)
register_PHI (....)
Orchestrator Selects Implementation
hotspot:
transfer data to accelerator
execute kernel(s)
transfer data to host
save output / post-processing
Heartbeat is incremented: provide information on Appl. performance

Figure 5.8: The heartbeat loop with the inserted accelerator hooks (registration functions).

Application Execution: Once the preparation phase (6) has been completed and the application has been modified to support the RTCS toolflow, the application can begin executing (7). The RTCS first runs further optimization passes on the LLVM IR and then initializes the LLVM EngineBuilder with the application module. It then JIT compiles the module, creates a new RTCS thread and starts executing the `main()` function through the LLVM Execution engine on this new thread.

5.9 Handling the Codegen Request

During the lifetime of the application, the Orchestrator monitors the application's performance via the heartbeat infrastructure. Depending on the application-level and system-level goals and defined policies, the Orchestrator can instruct the RTCS Client to generate code for a missing accelerator implementation ⑨. The RTCS Client adds the HDS to this code generation message and sends it to the RTCS Server ⑪. The codegen message consists of the HDS with a list of hotspot kernels identified in Section 5.7 along with the target accelerator. The RTCS Server uses this information to check if the code was previously generated and is available in its cache. If found in the cache, the RTCS Server skips the code generation steps and can immediately send the accelerated code back to the client. By using a cache and by serving all instances of the RTCS Client, the RTCS Server can enhance the reuse of already processed requests. If the accelerated code is not found in the cache, the RTCS Server proceeds with the code generation phase.

To generate corresponding accelerated code, the RTCS Server requires the sequential LLVM IR code for the hotspot (`CPUBaselineFunction`) as well as all the called kernel functions. The RTCS Server sends a message to the RTCS Client requesting for the LLVM IR of the corresponding hotspot and kernel(s). On receiving this request, the RTCS Client exports the appropriate LLVM IR by creating a new LLVM IR `module` and then creating copies of the hotspot and each kernel (function) into the newly created module. A `verification` pass is run on the module to check for any errors after which it is serialized and send to the RTCS Server.

In this thesis, we present a generalized OpenCL parallelization and code generation approach that is capable of generating OpenCL C kernel code from sequential code. However, different accelerator types have different properties and in turn different optimization methods. Specialized accelerator specific OpenCL code generators can be derived from the generalized OpenCL code generation approach to take into account these differences and further optimize the OpenCL kernel code. In the next section we look at how a sequential kernel is parallelized.

5.10 Work-Item Parallelizer

In a sequential application, work is performed consecutively in each loop iteration, with the loop iterator specifying which `work-item` is currently being executed. The work-item parallelizer ⑫ exposes parallelism in OpenCL by transforming data parallel loops into independent `work-items` that can be executed in parallel by the processing elements. Each OpenCL work-item executes the same kernel code, but works on different data as specified by the NDRange. Every work-item has a unique identifier which is used to identify what index the work-item needs to process. This identifier can be obtained by calling the `get_global_id()` OpenCL API call. The main task of the work-item parallelizer is to transform the identified sequential loops by replacing the loop iterator with appropriate calls to the `get_global_id()` API.

In Listing 5.3 we saw the pseudocode of a simplified 2D 3×3 convolution. The outer two loops (lines 2-3) iterate over the 2D output space. While the inner two loops (lines 6-7) perform a 3×3 convolution for each entry. During the analysis phase, the `DependenceAnalysis` pass (see Section 5.7.2) determines if the kernel can be parallelized and stores the maximum parallelization loop level in the HDS. By looking at this information in the HDS, the work-item parallelizer can determine that the outer two loops are parallelizable while the innermost loops are data dependent. For each parallelizable loop, the work-item parallelizer performs the following steps:

1. Determine the loop induction variable.
2. Remove the control flow statements of the loop.
3. Replace the induction variable with a call to the `get_global_id()` API call.
4. Store information about the loop iteration space that will be used in the host code to generate the appropriate work-items.

The induction variable of a loop is the variable that is incremented or decremented for each loop iteration. In our example in Listing 5.3, `r` and `c` are the induction variables of the outer loops. Since the LLVM bitcode is canonicalized (see Section 5.6), the loop induction variable is guaranteed to be represented as a `PHINode` instruction in the loop header. The `PHINode` belongs to every loop control flow and is used to select a value depending on the predecessor of the current basic block. Once the induction variable is found, the work-item parallelizer looks for the corresponding `ICmpInst` instruction that checks the loop exit condition (`r` $<$ `rows` -2). The loop-limit is stored in the HDS and is later used by the RTCS Client to invoke the OpenCL kernel with the appropriate work-items. The compare and branch instructions associated with the loop control flow are then removed. This effectively removes the loop structure with all the code previously inside the loop being executed exactly once. The final step is to replace the induction variable with a call to the `get_global_id()` OpenCL API call. Listing 5.4 shows the transformed pseudo code after all the steps are completed (loops in line 2-3 are replaced). It is important to note, that we use pseudo code to help the reader better understand the concepts, however the RTCS performs these code transformations at the LLVM IR level.

```
 1  // Iterate over the 2D output.
 2   int r = get_global_id(0);
 3   int c = get_global_id(1);
 4     // Convolve over the 3X3 filter.
 5     int sum = 0;
 6     for (int i = 0; i < 3; i++) {
 7       for (int j = 0; j < 3; j++) {
 8         // ...
 9         sum += in[r+i][c+j] * MASK[i][j];
10     // ...
11     out[r][c] = sum;
```

Listing 5.4: Pseudo OpenCL C code for a transformed parallelized OpenCL C kernel.

5.11 Tiling: Exploiting Local Memory

In the previous section we looked at how the work-item parallelizer transformed sequential loops into work-item based OpenCL C kernels. In our evaluation in Chapter 6, we will see that in most cases, this parallelization results in an increase in performance when compared to the original sequential application. However, this approach only makes use of the OpenCL device's global memory and as we saw in Section 4.2.1, accessing global memory is expensive. Some OpenCL devices utilize dedicated hardware units to help improve the global memory efficiency by combining consecutive work-item memory accesses together and issuing burst reads or writes. However, local memory accesses are still an order of magnitude faster than global memory, the only caveat being its limited size (see Figure 4.5). For applications with a lot of data reuse, one could further improve the parallel performance by making use of faster local memory.

Applications like matrix multiplication, stencils and convolutions are prime examples of applications with considerable data reuse. The RTCS makes use of a template-based approach to identify data reuse patterns in applications. Presently the RTCS can identify data reuse patterns for matrix multiplication and convolutions. However, this can be extended to support more applications in the future. To help understand how data reuse affects performance and how one can benefit by using local memory instead of global memory, we first take a look at the convolution filter in the next section.

5.11.1 Convolution

The convolution of two functions has a broad range of applications like image processing, signal filtering, audio processing and artificial intelligence. For example, in image processing applications, convolutions are used to sharpen the image, remove noise, detect edges and even blur images. A subset of convolutions known as separable convolutions are convolutions that can be separated out and represented as simpler convolutions which results in better performance. However, not all convolutions are separable. In our work, we develop a generalized tiling approach for convolution filters and hence do not consider such separable convolutions. 2-dimensional convolution filters form the core of many image processing algorithms, and can be defined as:

$$S'(x,y) = \sum_{i=-a}^{a} \sum_{j=-a}^{a} S(x+i, y+j) \cdot k(i+a, j+a) \tag{5.1}$$

where: S' is a 2-dimensional output image,
S is a 2-dimensional input image,
k is the $m \times m$ square convolution kernel being applied and
$a = \dfrac{m-1}{2}$, is the radius of the convolution kernel

Figure 5.9 shows an example of a 3×3 image filter. A pointwise scalar product of the source image and the kernel filter is performed to obtain the result matrix which is then summed up and stored in the output image. We can see that for each output computed, $m \times m$ input elements need to be loaded from the global memory and 1 output element stored to the global memory. This requires a high amount of memory bandwidth to process the entire image. Since the filter kernel is generally quite small, it can completely fit into the faster optimized constant global memory (see Section 4.2.1) and we do not consider it for optimization.

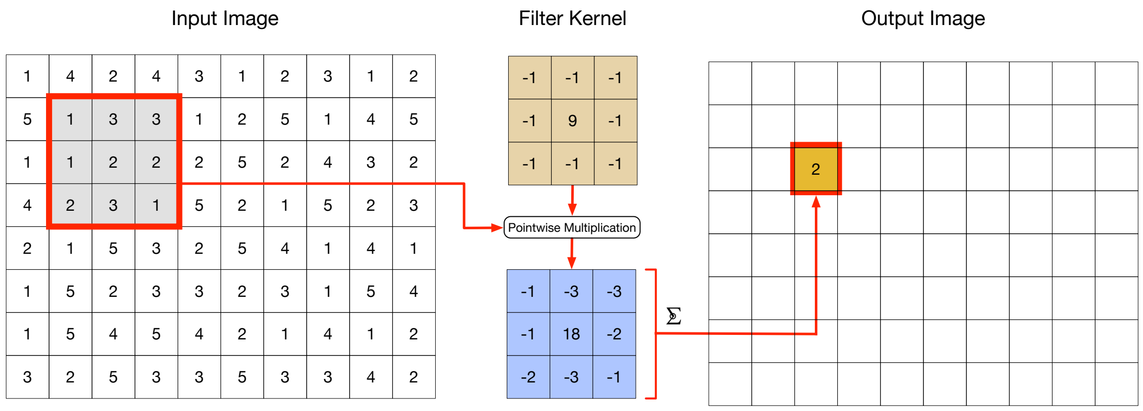


Figure 5.9: A basic 3×3 sharpness convolution filter.

For convolution filters, there is a significant amount of data reuse amongst neighboring elements. Figure 5.10 demonstrates the data reuse amongst two neighbors for a 3×3 convolution filter. The highlighted cells show the data overlap between the two neighboring computations. We can see that only 3 elements differ between the data requirement of the first and the second computation. Since the access pattern is constant for a given filter size, we can take advantage of the faster local memory by first copying the data from the global memory to the local memory of the compute unit and then performing the computation. However, as the local memory size is limited, the iteration space is partitioned into smaller tiles that can fit in local memory. This method is known as *Tiling* and is typically used in CPUs to help maximize the number of cache hits, thereby reducing the cost to fetch data from other slower cache levels or from the main memory.

In OpenCL, local memory can only be shared between work-items belonging to the same work-group and hence the iteration space is partitioned/tiled so that each tile is mapped to a work-group. Figure 5.11 demonstrates how a 2-dimensional output space can be partitioned into different tiles. Each tile is processed by a single work-group and the bold green squares in the figure denote the work-groups (tiles) while the small squares denote individual work-items.

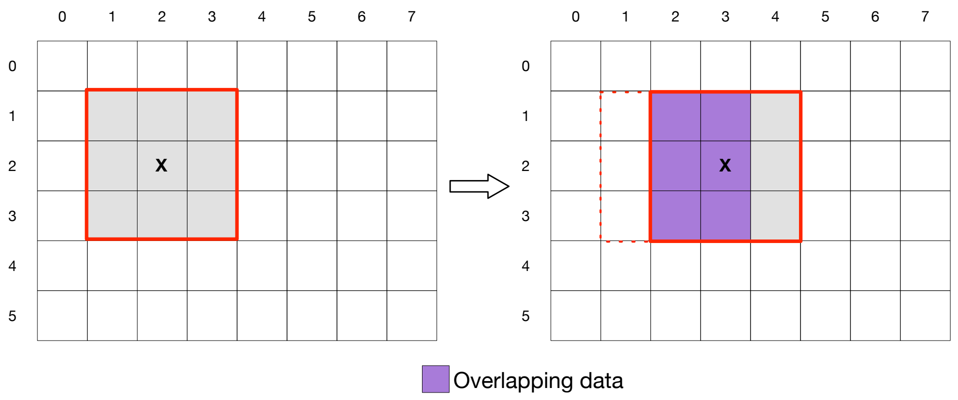


Figure 5.10: Data reuse between two neighbors for a 3×3 convolution filter.

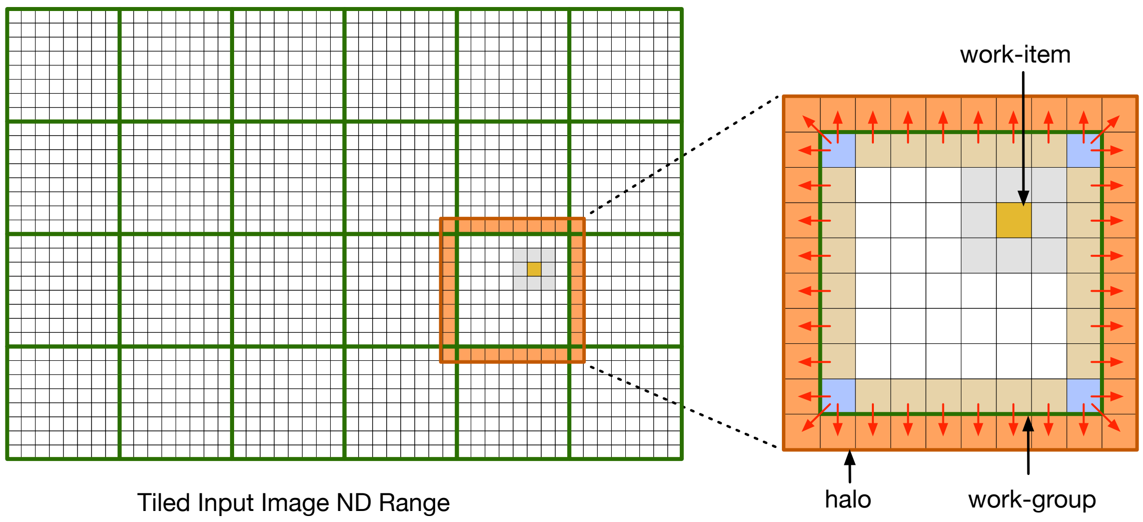


Figure 5.11: Loading the required halo region of a work-group.

Transferring data to local memory: When using the tiling approach, each work-item initially loads one element from global memory into local memory. This results in the entire contents of the tile being loaded into local memory. However, because of the data access pattern of a convolution filter, the work-group needs additional data (borders) to perform all its computations. This is shown in Figure 5.11 as the orange elements and is known as the *halo* region. The size of this halo region depends on the radius of the convolution filter. The radius is computed as $(filter\_size - 1)/2$. In our example, the $filter\_size$ of the 3×3 convolution filter is 3 and the radius is 1. This means that we have to load a halo region of size 1 around the tile. In our implementation, this radius is used to determine which work-items are used to load the extra halo elements into local memory. The approach is described in Algorithm 1. In our example in Figure 5.11, the radius of the halo is 1 and hence only the adjacent bordering elements are used. The red arrows in Figure 5.11 show which additional global memory elements are loaded by a work-item. We see that a bordering work-item loads one additional element except for the corners where three items are loaded.

Algorithm 1: Algorithm to load the halo region of a work-group/tile into local memory for a convolution filter.

```
r = (filter_size - 1)/2;
x = get_local_id(0);
y = get_local_id(1);
wgs = WORK_GROUP_SIZE;
if x <r then
    //Load left halo
    load_local_mem( x - r , y);
if x >= wgs - r then
    //Load right halo
    load_local_mem( x + r , y);
if y <r then
    if x <r then
        //Load the top left corner
        load_local_mem( x - r , y - r);
    //Load top halo
    load_local_mem( x , y - r);
    if x >= wgs - r then
        //Load the top right corner
        load_local_mem( x + r , y - r);
if y >= wgs - r then
    if x <r then
        //Load bottom left corner
        load_local_mem( x - r , y + r);
    //Load bottom halo
    load_local_mem( x , y + r);
    if x >= wgs - r then
        // Load the bottom right corner
        load_local_mem( x + r , y + r);
```

For the sake of simplicity and to help the reader better understand Algorithm 1, the global to local memory mapping as well as out-of-bounds checks have been abstracted away in the `load_local_mem` function. Additionally, Figure 5.11 as well as Algorithm 1 only consider a symmetric convolution where the convolution kernel is symmetric across its mid point. However, convolutions like the one shown in Listing 5.3 are skewed i.e. the

3×3 convolution loop in lines 6 and 7 do not run from -1 to 1 but instead run from 0 to 2. The same amount of data needs to be loaded as in the symmetric case, however with a different memory offset. This can be addressed by adding a `SKEW_FACTOR` that offsets the global data loaded by the `load_local_mem` function.

Synchronization In Section 4.2 we saw that a work-group is executed on a compute unit in the form of wave-fronts. All work-items within a single wave-front are executed in parallel in lockstep. However, different wave-fronts within a single work-group are executed sequentially. Since each work-item depends on the neighboring work-items to load the required data into local memory, it is possible that the neighboring data has not yet been loaded into local memory when a work-item starts its computation. To avoid this, a synchronization point in the form of a *barrier* is used to guarantee that individual work-items can only proceed with subsequent computations after all work-items in a work-group have reached the synchronization point and, in our scenario, all the required data has been transferred to local memory.

The Tiling Pass The RTCS Server uses our custom `tiling` pass ⑬ that first analyses the hotspot kernel for a convolution pattern and then tiles the hotspot kernel by transforming the LLVM IR. Listing 5.5 shows the pseudo OpenCL C code for the tiled convolution filter initially parallelized in Listing 5.4. To tile a convolution filter, the `tiling` pass performs the following steps:

1. Determine if a convolution is present
2. Identify the filter and its size
3. Create a local buffer based on the work-group size and filter size
4. Insert code to transfer data (along with the halo) from global to local memory
5. Add a synchronization point (barrier) for local memory
6. Replace global variables and iterators with the corresponding local buffers and iterators

The `tiling` pass first analyses the data access patterns of the hotspot kernel to determine if a convolution computation is present. It makes use of a pattern matching approach that looks at the load and store patterns inside loops as well as the corresponding computation (reduction) to identify a convolution pattern. If a convolution pattern is identified, its filter size and radius are computed and the `tiling` pass proceeds with the remaining steps.

Local memory buffers are required to hold the corresponding tiled data from global memory. During the analysis phase, the `tiling` pass analyses the data access patterns and determines which data-arrays require local storage. If we look at the example in Listing 5.4, we can see that local memory is required for the `in` data-array. The size of the local memory that is required to be allocated depends on the filter radius computed in the previous step and the size of the local work-group (Listing 5.5, line 9). Since we map a tile to a work-group, this is the same as the tile size. The local work-group size is obtained from the HDS and is previously initialized by the RTCS Client. This size can be manually specified as a command line parameter to the RTCS, otherwise a default size of 16×16 is used.

Once the local data-array has been created, the code required to load the required data from global memory to local memory is inserted into the hotspot kernel according to Algorithm 1 (Listing 5.5, line 11). A synchronization barrier is inserted after this step to ensure that all the required local memory has been loaded (line 14). All global data-array accesses inside the convolution loop are replaced by corresponding local data-array accesses and their indexes are updated to use the local iterator instead of the global iterator (line 21). Finally, the HDS is updated to indicate that the kernel was tiled. This information is used later by the RTCS Client to invoke the OpenCL kernel with the appropriate global and local work-group parameters.

```
 1  // Iterate through 2D input.
 2   int r = get_global_id(0);
 3   int c = get_global_id(1);
 4
 5   int local_r =  get_local_id(0);
 6   int local_c =  get_local_id(1);
 7
 8   // Allocate local memory for the tile
 9   __local int local_in[(WGS + FILTER_RADIUS*2)*(WGS + FILTER_RADIUS*2)];
10
11   copy_to_local(in, local_in, r, c, local_r, local_c);
12
13   //Guarantee that the complete tile is loaded into local memory
14   barrier(CLK_LOCAL_MEM_FENCE);
15
16   // Convolve over the 3X3 filter.
17   int sum = 0;
18   for (int i = 0; i < 3; i++) {
19     for (int j = 0; j < 3; j++) {
20       // ...
21       sum += local_in[local_r+i][local_c+j] * MASK[i][j];
22      // ...
23   out[r][c] = sum;
```

Listing 5.5: Pseudo OpenCL C code for a tiled convolution kernel (skewed) that uses local memory. `WGS` denotes the work-group size and the value of `FILTER_RADIUS` is 1 in this case.

To evaluate the theoretical benefits of using tiling, we compare the number of global memory loads required per tile for the tiled and non-tiled approaches. It is important to note that we use a "tile" only as a unit of comparison between the tiled and non-tiled approaches. To compute this, we use square tiles as well as square filters, with a size of $tile\_size \times tile\_size$ and $filter\_size \times filter\_size$ respectively. We first look at the global memory accesses required per tile when only global memory is used in the non-tiled approach. Each element in the tile requires $filter\_size \times filter\_size$ reads from global memory. This results in $tile\_size^2 \times filter\_size^2$ global loads per tile. By taking advantage of tiling and using local memory instead of only global memory, the number of loads from the global memory loads are computed by:

$$\begin{aligned} global\_memory\_loads &= \#elements\_per\_tile + \#halo\_elements \\ &= tile\_size^2 + \left(tile\_size + \frac{filter\_size - 1}{2}\right) \times 4 \end{aligned} \tag{5.2}$$

Compared to the non-tiled approach, we can see that the number of global memory loads is considerably reduced, allowing the application to benefit from faster local memory. In the next section, we take a look at the matrix multiplication application and how it can be tiled.

5.11.2 Matrix Multiplication

Matrix multiplication is widely used in diverse domains like graph theory, statistics, engineering and computer graphics, including image transformation like stretching, squeezing, rotation, scaling, sheering or reflection. Given that we want to multiply two input matrices A of size $m \times n$ and B of size $n \times p$, each element c of the resulting matrix C $(m \times p)$ is computed by:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj} \tag{5.3}$$

$$\text{where:} \quad i = 0, ..., m-1 \text{ and} \\ j = 0, ..., p-1$$

Figure 5.12 visually depicts how the resulting matrix element is computed from the two inputs matrices A and B. The element c_{ij} in the result matrix C is obtained by first computing the pairwise multiplication of n elements of the i^{th} row of A and the j^{th} column of B and then summing up the resulting n elements. Applying our parallelization approach, the RTCS is able to parallelize the outer two loops. Listing 5.6 shows the pseudo OpenCL C code that is generated by the RTCS.

```
int i = get_global_id(0);
int j = get_global_id(1);
int tmp = 0;
//Matrix multiplication
for (int k = 0; k < n; k++) {
  tmp += A[i][k] * B[k][j];
}
C[i][j] = tmp;
```

Listing 5.6: Pseudo OpenCL C code for a matrix multiplication kernel using global memory.

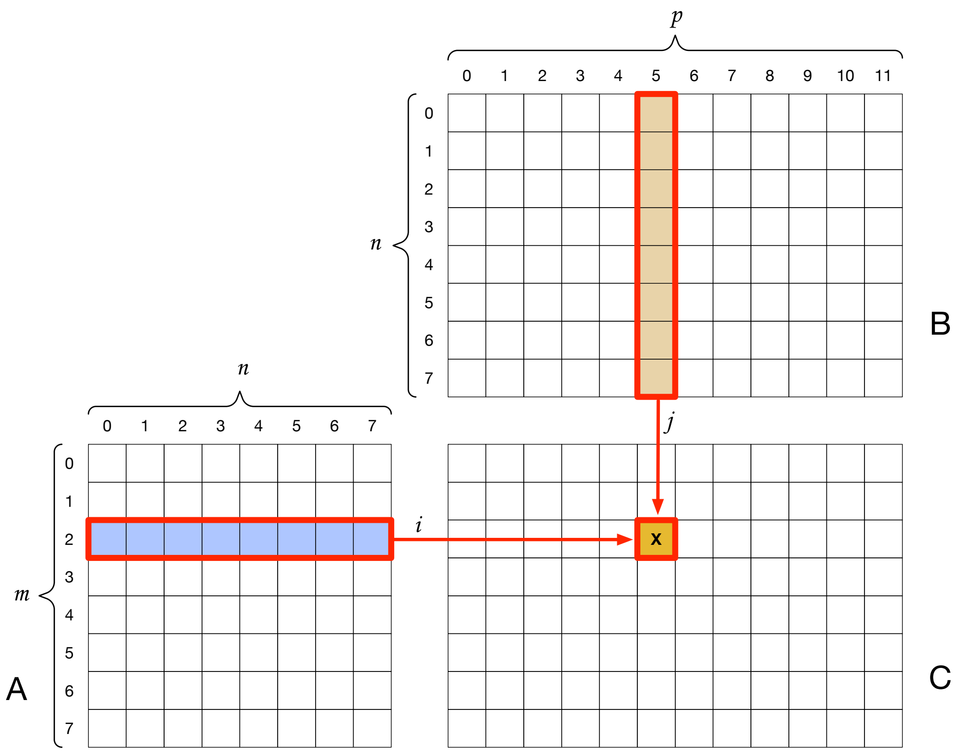


Figure 5.12: Computation of element C_{ij} for $C = A \times B$.

Looking at Figure 5.12, we can see that in order to compute a value, a complete row from matrix A and a complete column from matrix B needs to be fetched. This means that $2 \times n$ elements need to be loaded from global memory to calculate a single element in matrix C, resulting in $2 \times m \times n \times p$ loads from global memory for the complete matrix. However, we can also see that there is a lot of potential for data reuse. For every row to be computed in matrix C, the same row from matrix A can be reused (blue area in the figure). Similarly, for every column to be computed in matrix C, the same column from matrix B can be reused (light brown area in the figure). To help reduce global memory access and to take advantage of data reuse, we apply a tiling approach based on the one presented by Rivera et al. [71]. The matrix is first divided into multiple tiles or sub-matrices (see Figure 5.13). A partial tile-wise computation is performed and all the partial results are summed up to calculate the final result. If we partition A into $q \times r$ tiles and B into $r \times s$ tiles, the resulting tile in C is computed by:

$$C_{ij} = \sum_{k=0}^{r-1} A_{ik} \times B_{kj} \tag{5.4}$$

$$\text{where:} \quad i = 0, ..., q-1 \text{ and} \\ j = 0, ..., s-1$$

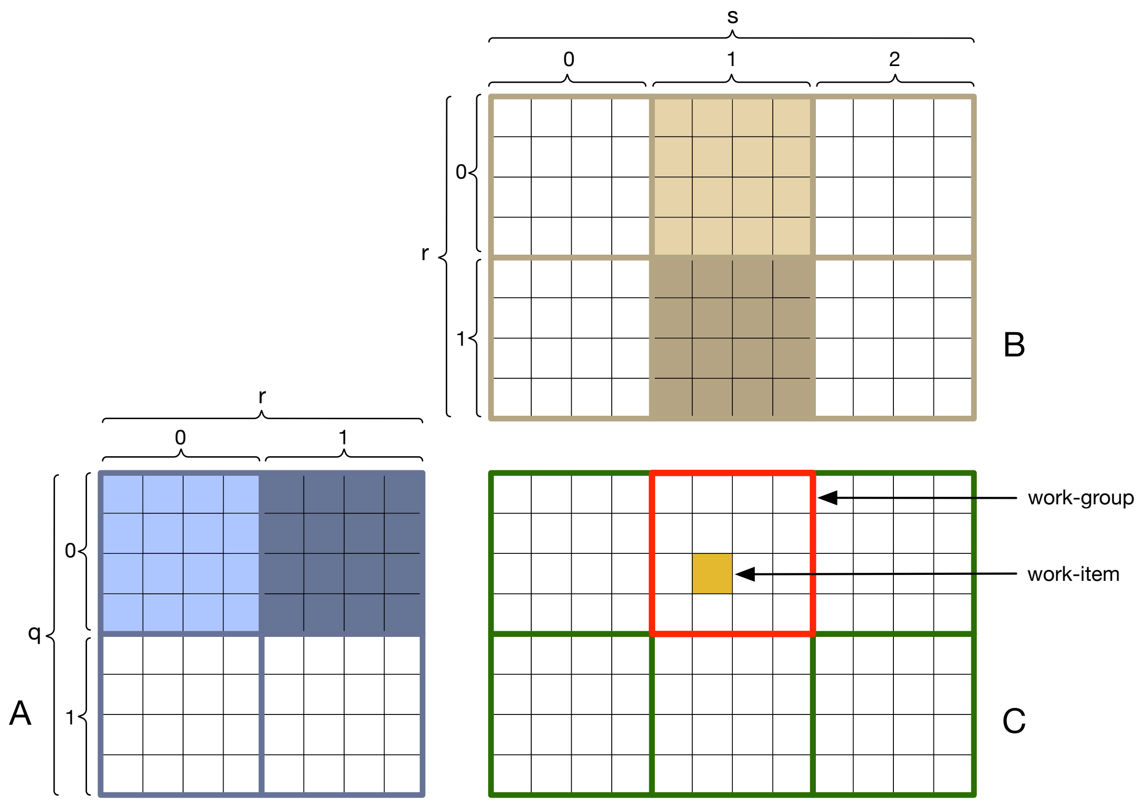


Figure 5.13: Tiled Matrix Multiplication.

Figure 5.13 shows how such a tiled approach for the matrix in Figure 5.12 looks like. The matrices have been partitioned into tiles of 4×4 elements and each tile in matrix C forms its own work-group. To compute the tile C_{01}, initially tiles A_{00} and B_{01} are loaded into local memory. The matrix multiplication of $A_{00} \times B_{01}$ is performed and stored locally. Tiles A_{01} and B_{11} are then loaded into local memory and the matrix multiplication is repeated. The result is then summed up and finally stored in C_{01}. Since all the work-items share the same local memory, every work-item loads only one element from global memory for every input matrix (A, B) resulting in fewer memory access to global memory. In this example we use square tiles each of size $tile\_size \times tile\_size$. To compute the resulting tile of matrix C, the work-group needs to load $tile\_size \times n$ elements from matrix A and $n \times tile\_size$ elements from matrix B, resulting in $2 \times n \times tile\_size$ loads from global memory. The total number of global memory loads required to compute matrix C using square tiles is given by:

$$global\_memory\_loads = 2 \times n \times tile\_size \times \frac{m}{tile\_size} \times \frac{p}{tile\_size} = \frac{2 \times n \times m \times p}{tile\_size} \quad (5.5)$$

When compared to the non-tiled approach, the number of global memory loads is reduced by a factor of $tile\_size$.

The `tiling` pass (13) of the RTCS Server analyses the hotspot kernel to detect a matrix multiplication pattern. It does so by using the LLVM `LoopInfoPass` to examine the structure of the loops. It also looks at load and store operations as well as the computations inside the loop in order to identify a matrix multiplication pattern. If a matrix multiplication pattern is found, the data-arrays involved are identified and the order of the multiplication is determined ($A \cdot B \neq B \cdot A$) after which the `tiling` pass transforms the main multiplication loop structure. Listing 5.7 shows the tiled pseudo OpenCL C code based on the code in Listing 5.6. The variables from Figures 5.12 and 5.13 are also reused to better understand how the matrix multiplication kernel is tiled by performing the following steps.

1. Add code to compute the number of tiles required. ($tiles = n/tile\_size$)
2. Create local sub-matrices to store the input tiles. ($local\_A$, $local\_B$)
3. Encapsulate the matrix multiplication loop with the tiling loop. (Listing 5.7: line 15)
4. Compute the tiled indices for global memory accesses.
5. Add code to transfer data from global to local memory.
6. Insert a synchronization point (barrier) for local memory.
7. Update the matrix loop to iterate up to $tile\_size$ instead of n
8. Translate addressing and memory accesses inside the matrix loop from global to local memory.
9. Insert a barrier to wait for all work-items to finish before proceeding to the next tile.

Presently, the RTCS only supports square tiles with the $tile\_size$ being obtained from the HDS, which is previously initialized by the RTCS Client. This size can be manually specified as a command line parameter to the RTCS, otherwise a default size of 16×16 is used. Similar to the tiling approach for convolutions, local data-arrays are created to hold the tiles required for the input matrices (A and B), and their sizes are set to the $tile\_size$ (Listing 5.7, lines 11 and 12).

To compute one tile in the resulting matrix C, one needs to explicitly iterate over different row-tiles from matrix A as well as the corresponding column-tiles from matrix B. The `tiling` pass needs to introduce an additional loop to iterate over these tiles. The `tiling` pass inserts a new tiling loop that iterates over the number of (row/column) tiles (Listing 5.7, line 15) into the hotspot kernel, encapsulating the matrix multiplication loop. It also introduces new array indices to correctly address the local and global data-arrays. The local data-arrays are then populated. A synchronization barrier is inserted after this step to ensure that all the required data has been loaded into local memory (line 23).

```
int i = get_global_id(0);
int j = get_global_id(1);
int tmp = 0;

int tiles = n / tile_size;

int local_i = get_local_id(0);
int local_j = get_local_id(1);

// Allocate local memory for the tiles
__local float local_A[tile_size][tile_size];
__local float local_B[tile_size][tile_size];

//The tiling loop
for (int tileIter=0; tile<tiles; tile++) {
  // Load one tile of A and B into local memory
  int tiled_i = tile_size*tileIter + local_i;
  int tiled_j = tile_size*tileIter + local_j;
  local_A[local_i][local_j] = A[i][tiled_j];
  local_B[local_i][local_j] = B[tiled_i][j];

  //Guarantee that the complete tile is loaded into local memory
  barrier(CLK_LOCAL_MEM_FENCE);

  // Matrix  multiplication per tile (main MM loop)
  for (int k=0; k<tile_size; k++) {
    tmp += local_A[local_i][k] * local_B[k][local_j];
  }

  //Wait for all work-items to finish before loading the next tile
  barrier(CLK_LOCAL_MEM_FENCE);
}
C[i][j] = tmp;
```

Listing 5.7: Pseudo OpenCL C code for a matrix multiplication kernel using tiling and local memory.

5.12 Generating and compiling OpenCL Kernel Code

After the hotspot kernels have been parallelized and tiled, the hotspot kernels are translated from LLVM IR to OpenCL C Kernels (14). We rely on Axtor [65], an open-source AST-extractor for LLVM to generate OpenCL C code. Axtor takes LLVM IR as input, restructures the control flow to suit high-level code generation and then generates OpenCL C code from it. Axtor can be considered to be a back-end for LLVM, which can translate LLVM bitcode into OpenCL C code. Unfortunately, the initial efforts were not continued and the source code has been removed from the LLVM tree due to lack of maintenance. Magni et al. [63] revived and adapted Axtor to LLVM 3.5 for their thread-coarsening approach. We used this version as a base and ported it to LLVM 3.8.0. We also extended

Axtor to support the generation of address space qualifiers for local and constant variables.

In our approach, the RTCS Server uses the Axtor back-end to generate OpenCL C kernel code for each hotspot kernel separately. Once all the OpenCL C kernels have been generated, they are consolidated into a unified `*.cl` file. The server then uses the `clCreateProgramWithSource()` OpenCL API call to compile the OpenCL kernel into a OpenCL program for the specified OpenCL device. It then uses the `clGetProgramInfo()` OpenCL API call to extract the device specific binary from the compiled OpenCL program and saves it to file ⑮. The RTCS Server cache is updated and the OpenCL binary along with the updated HDS is sent to the RTCS Client. In the next section, we take a closer look at how the corresponding OpenCL host code is generated by the RTCS Client and how the accelerated code is integrated into the application.

5.13 Integrating Accelerated Code into the Host

The generated parallel OpenCL kernel code represents only the raw computation on a compute device. A kernel executes on multiple processing elements in parallel and corresponding OpenCL host code is required to setup the device, load the kernel code, handle data buffers and enqueue the kernel. In this section, we describe how the RTCS Client automatically generates the OpenCL host code and integrates it into the application. We start by looking at the different steps that need to be performed during a typical execution of an OpenCL application.

1. Select the appropriate platform and device
2. Create a device context and command-queue
3. Loading (or compiling) the OpenCL program and creating kernel handles
4. Create input/output data buffers on the device
5. Transfer data from the host to the device
6. Set the kernel arguments
7. Execute the kernel(s) on the device (enqueue)
8. Transfer data back to the host

An OpenCL platform is a vendor specific implementation of the OpenCL specification and is used by the host to control different OpenCL devices belonging to it. The OpenCL runtime is capable of supporting multiple OpenCL platforms, where platforms from different vendors can coexist on the same host. As an example, the heterogeneous node used as our evaluation platform supports the Intel as well as the Nvidia OpenCL platforms. Each platform contains one or more OpenCL devices (e.g. mCPU, MIC) which perform the actual computation. The OpenCL runtime uses a context to manage command-queues, memory, program and kernel objects [49]. A context may have one or more devices, with each device within a context having its own command-queue. All commands intended for the device (buffer creation, data transfer, kernel execution, synchronization, etc) are submitted to the command-queue. By default, all the commands in the command-queue are executed in-order, allowing it to be used to specify the execution order. In our approach, the code required to select the appropriate platform/device and create the command queue (steps 1-2) are previously inserted into the application during the application preparation phase (see Section 5.8).

5.13.1 Loading the OpenCL Program

On receiving the OpenCL code (binary) from the RTCS Server, the RTCS Client first saves it locally. To execute the kernel(s) on an OpenCL device, the OpenCL runtime needs to first load this device specific binary. This step needs to be performed only once during the application's lifetime. This is similar to registration function which also needs to be executed only once within the application. The RTCS exploits this and uses the appropriate registration function (created during the application preparation phase) to also load the pre-compiled OpenCL binary by inserting the OpenCL API call that first loads and then builds the OpenCL program (step 3). Creating OpenCL kernel handles also need to be performed only once during the application's lifetime and the RTCS also inserts code that creates OpenCL kernel handles for each of the hotspot kernels into the registration function. This registration function is called within the heartbeat loop when the application is executing and has been previously discussed in Section 5.8.

5.13.2 Generating OpenCL Host Code

OpenCL host code is also required to setup the data buffers, transfer the required data to the device, enqueue the kernel and transfer the data back to the host. To simplify the OpenCL host code generation, the RTCS Client first creates a new empty function with the same signature as the `CPUBaselineFunction`. All the OpenCL commands to transfer data and launch the OpenCL kernels are then added to this new function. The main objective of this new function is to provide a container for steps 5-8 and is known as the accelerated function.

However, before the data can be transferred, the RTCS needs to allocate data buffers of the appropriate data type, size and read/write type on the OpenCL device. The data type, buffer type and size are obtained from the HDS, which were previously populated during the data analysis phase described in Section 5.7.3. The OpenCL data buffers are created on the device by calling the `clCreateBuffer()` API (step 4). The `clEnqueueWriteBuffer()` API call is then used to transfer data from host memory to the data buffer on the OpenCL device (step 5).

The kernel arguments need to be set before the kernel is invoked and this is done by adding the `clSetKernelArg()` API call to specify the argument index and the corresponding data (steps 6). The arguments are either pointers to a device buffer previously created (in the case of data-arrays) or basic data-types which are obtained from the HDS. Once the arguments are set, the kernel is enqueued into the command queue by calling the `clEnqueueNDRangeKernel()` API which is used to specify which kernel to execute along with the total number of work-items and the work-group size. In addition to the OpenCL kernel code, the RTCS Client also receives an updated HDS from the RTCS Server. For every OpenCL kernel, the updated HDS contains information about the work dimension and the global and local work-group sizes which are used by the RTCS Client to launch the OpenCL kernel with the appropriate parameters (step 7). The OpenCL kernels are invoked in the same order as specified in the hotspot.

Data is transferred back to the host by using the `clEnqueueReadBuffer()` API call (step 8). A `clFinish()` command is inserted into the device command queue to ensure that all the data is written to the host before finally cleaning up and releasing the buffers on the device using the `clReleaseMemObject()` API call.

As a further optimization, the data transfer steps are skipped if the OpenCL device is the `mCPU`. In this case, the host and the OpenCL device share the same (host) memory. The data buffers on the device are created using the `CL_MEM_USE_HOST_PTR` flag along with a pointer to the host memory which tells the OpenCL runtime to use host memory.

5.13.3 Registering the new Accelerator Implementation

At this point, the OpenCL host code has been generated and integrated into the application, however before the Orchestrator can take advantage of the new accelerator implementation, it needs to be registered with the Orchestrator. Additionally, the application also needs to know how to call this new implementation. In a SAVE-Enabled application, each hotspot implementation is called within a lambda expression. The lambda expression serves as a container which defines the accelerator specific function to be called as well as all the required arguments. This lambda expression along with the accelerator type is then registered with the Orchestrator (see Section 5.6). At runtime, when the Orchestrator selects an implementation that the application should use, the application checks the accelerator type and executes the corresponding lambda expression.

To register the new accelerator with the Orchestrator, the RTCS needs to first create a lambda expression that calls the accelerated function described in the previous section. Internally, a lambda expression is represented as a class and generating such a lambda expression using the LLVM API is not straightforward. The LLVM C back-end is used to aid in generating the lambda expression. LLVM provided an inbuilt C back-end until version 3.1, however it has since been discontinued due to lack of maintenance. Fortunately, the Intel `ispc` project [25] is an LLVM based compiler which also needs to transform LLVM IR to C code in order to use it with the Intel Compilers. They revived and adapted the LLVM C back-end for internal use and have released it as an open-source project. We make use of this back-end in our project. Only minor modifications have been made to resolve dependencies to the rest of the `ispc` project and integrate it cleanly. The result of these modifications is an independent software module which provides an interface to generate a `*.c` file from LLVM IR code.

The RTCS makes use of this LLVM C back-end to generate the C function signature for the accelerated function. Using the generated C code, the RTCS constructs a C++ file with a lambda expression with a call to the accelerated function. The function signature is then inserted into this lambda template and compiled using *clang* to obtain the corresponding lambda expression in LLVM IR for the newly created accelerated function.

In Section 5.8, we saw that accelerator hooks in the form of registration functions were added to the application to help the RTCS register new accelerator implementations with the Orchestrator at application runtime. The `registerImplementation()` Orchestrator API call, with the lambda expression and accelerator type as arguments is inserted into the appropriate registration function. The registration function is called for every iteration of the heartbeat loop and to ensure that the registration function is only executed once, a global boolean variable is added to the application and initialized to `false`. The registration function first checks this variable and only loads the kernels and registers the implementation if it is set to `false`. Once the registration function executes, this value is set to `true`. Finally, the updated registration function is Just-in-Time compiled using the LLVM Execution Engine and the registration function pointer is updated to point to the newly compiled registration function. During the next iteration of the heartbeat loop, the new accelerated implementation is registered with the Orchestrator. The Orchestrator now has the possibility to select the new accelerated implementation.

5.14 Chapter Conclusion

In this chapter, we took an in-depth look at the Runtime and Just-in-Time Compilation System and its different components. We saw how the RTCS supplements the Orchestrator by JIT generating missing accelerator implementations for SAVE-Enabled applications. To help streamline accelerator code generation, the RTCS was separated into two main components, the RTCS Client and the RTCS Server. A communication module was developed to allow the Client and Server to efficiently interact with one another and also communicate with the Orchestrator. We looked at the RTCS toolflow in detail and saw how the application was loaded into the RTCS, canonicalized and analyzed for Orchestrator constructs. The different custom LLVM analysis passes like the `ExternalFunctionFilter`, `DependenceAnalysis` and `DataOptimization` analysis passes used to analyze the hotspot were presented. We also saw how the application was prepared by adding hooks for missing accelerator implementations and how the application was executed within the LLVM execution engine. The different phases of OpenCL code generation were also presented in detail. We saw how the sequential kernel code was parallelized and how the generated OpenCL code was tiled using two different tiling strategies. The method to translate LLVM IR into OpenCL C, as well as the process of compiling OpenCL kernels into device specific binaries was presented. We finally saw how the OpenCL host code was generated, integrated into the application and registered with the Orchestrator. In the next chapter, we present an in-depth evaluation of our approach.

CHAPTER 6

Evaluation

In this chapter, we present a detailed evaluation of our approach. We first present the hardware platform, the benchmark applications and the measuring method. We analyze the overheads introduced by the RTCS in Section 6.4 as well as evaluate the improvement in application performance in Section 6.5. To give additional valuable insights on the impact of the RTCS, we present the results at the kernel as well as the hotspot level.

6.1 The Heterogeneous Evaluation Platform

The evaluation is performed on a multi-accelerator heterogeneous platform built using off-the-shelf hardware components. The heterogeneous platform runs CentOS 6.8 Linux with kernel v2.6.32. The heterogeneous node consists of a dual socket Dell PowerEdge T620 server with two Intel Xeon E5-2609 v2 CPUs as host processors, each with four physical cores (without simultaneous multithreading) and 32 GiB of main memory. The platform also features two additional accelerators with distinct architectures to offload the computation: an Nvidia Tesla K20c GPGPU and an Intel Xeon PHI 31S1P — both connected via PCI Express. Table 6.1 shows the distinct features of each device. The CPUs have 8 physical cores and are able to execute 8 threads in parallel at the same time. The GPGPU has 13 multiprocessors with 192 CUDA Cores per multiprocessor resulting in 2496 CUDA cores. While the Xeon PHI has 57 cores with four hardware threads per core.

OpenCL Devices: All the heterogeneous devices support OpenCL version 1.2. From now on we refer to the OpenCL devices corresponding to the Intel Xeon CPUs, Nvidia Tesla GPGPU and the Intel Xeon PHI as the `mCPU`, `GPGPU` and `PHI` respectively. The OpenCL runtime reports the number of OpenCL Compute Units as 8, 13 and 224 for the `mCPU`, `GPGPU` and `PHI` respectively. However, the number of Compute Units is not a clear indicator of the overall performance capability of the device. Each Compute Unit executes multiple threads (work-items) at the same time. For example, the Compute Units of the `GPGPU` can use up to 2496 threads (under optimal conditions: no thread-divergence and maximal occupancy). Whereas, the Compute Units of the `PHI` can execute only 224 threads, but each of them can execute different instructions (i.e. they support the thread-divergence).

In the next section we introduce all the benchmark applications and look at what each application does.

Table 6.1: The specifications of the different heterogeneous devices.

Compute device	Intel Xeon E5-2609 v2 (×2)	Nvidia Tesla K20c	Intel Xeon PHI 31S1P
Core frequency	2500 MHz	758 MHz	1100 MHz
# of cores	4 (×2)	13 (multiprocessors)	57
# of hardware threads	4 (×2)	2496 (CUDA cores)	228
memory size	32 GiB	5 GiB	8 GiB
cache L1 size	4×32 KiB[3] (×2)	13×48 KiB[4] (max.)	57×32 KiB[3]
cache L2 size	4×256 KiB[3] (×2)	1.5 MiB	57×512 KiB[3]
cache L3 size	10 MiB[5] (×2)	–	–
microarchitecture	Ivy Bridge	Kepler	Knights Corner
manufacturing process	22 nm	28 nm	22 nm
thermal design power	80 (×2) W	225 W	270 W

6.2 Benchmark Applications

We use a diverse set of benchmark applications to evaluate our approach. The applications are extracted from a broad set of domains (scientific computing, signal and image processing, security, etc.) Table 6.2 lists all the applications by name and a small description. The benchmark applications are single-threaded CPU applications written in C that have been adapted to be SAVE-Enabled applications (SEAs) and use heartbeats. In this section we take a look at these benchmark applications.

6.2.1 Dense Matrix Multiplication

The `2mm` application is a 2-dimensional dense matrix application that multiplies two matrices together and is commonly used in scientific and graphics domains. In every heartbeat iteration, the application multiplies two matrices. Computing an element of the resulting

[3] 8-way set associative cache.
[4] A 64 KiB block of memory is split between the L1 cache and shared memory.
[5] 20-way set associative shared cache.

Table 6.2: Benchmark applications and their description.

Application	Description
2mm [70]	2D dense matrix multiplications
bsop [85]	Black-Scholes option pricing for European options
fir [67]	Finite impulse response signal processing
enhance [24]	Convolution with a Gaussian function
heat2D [70]	heat2D equation solver
nbody [37]	N-body particle simulation (kernel)
motion [41]	Motion detection (contains six kernels)
raytrace[82]	Renders an 2D image
sha256[27]	Cryptographic hash function with 256 digest
stereo2D [Vaz14]	2D stereo matching

matrix is not dependent on the computation of any other elements and can be parallelized by the RTCS. The RTCS can also identify the matrix multiplication pattern and applies tiling optimizations.

6.2.2 Black-Scholes Option Pricing

The Black-Scholes Option Pricing application is a financial options pricer application based on the Black-Scholes formula [11], which is a method used to measure the risk of loss on a portfolio of financial assets. The `bsop` application is a simplified representation of the type of processing used by commercial financial risk analysis products. It takes a time in the future (the horizon time), a set of underlying instruments (e.g. stocks) and a portfolio as inputs. Using these inputs, `bsop` computes the financial risk or Value at risk (VaR) over many thousands of market scenarios. In every heartbeat iteration, the `bsop` application computes (or recomputes) VaRs for different horizon times, stocks and portfolios. Computing the VaR for a given market scenario is independent and can be parallelized by the RTCS.

6.2.3 Finite Impulse Response

Finite Impulse Response (FIR) is commonly used to filter signals in digital signal processing applications. FIR can be used for different kinds of DSP filters, like low-pass, high-pass, band-pass or band-stop filters. The `fir` application uses a band-pass filter. In every heartbeat iteration, the application processes a different signal. The input signal is represented as an array ($x[n]$) and every element of the output signal can be computed in parallel and is parallelized by the RTCS.

6.2.4 Image Sharpening

The `enhance` application is an image processing application that uses a convolution filter (9×9) to sharpen the input image. The input image is composed of three channels (red, green, blue). In every heartbeat iteration, the enhance application processes a different image. Since every output pixel can be independently computed, the RTCS is able to parallelize this application. The RTCS also identifies the convolution pattern and applies tiling optimizations.

6.2.5 Heat Transfer Simulation

The `heat2D` application performs a simulation of the transfer of heat in a 2-dimensional material. Every iteration of the heartbeat loop represents a time-step in this simulation. However, within a time-step, there are no dependencies and all the elements belonging to the 2-dimensional material plane can be computed in parallel. Internally, the computation of heat2D is represented as a 5×5 convolution filter and is tiled by the RTCS.

6.2.6 N-body Simulation

The `nbody` application performs the N-body simulation of the motion of particles and their interaction with one another. N-body simulations are widely used in physics, especially in astrophysics to study how different physical forces (e.g. gravity) affects the motion of different particles (e.g. planets, stars, etc.) in the universe. Every iteration of the heartbeat loop represents a time-step in the simulation. In each time-step, every element (body) calculates its new position based on the collective forces being applied to it by all other bodies in this simulation. This results in two loops, an outer loop that iterates over all bodies in the simulation and an inner loop to compute the collective forces being applied on that body by all other bodies. The position of the body is updated in the outer loop after the forces have been computed. This results in a dependency between the inner and the outer loop. The RTCS detects this dependency and only parallelizes the outer loop.

6.2.7 Motion Detection

The `motion` application is a collection of image processing kernels that takes two images as inputs, detects the motion between them and highlights it in the output image. The application makes use of six different kernels and is depicted in Figure 6.1. Both input images are first processed using a `gauss` filter to blur the image. The images are then converted from the RGB color space to the gray color space (`rgb2gray`). The results are fed into the `motion` detection kernel which detects motion depending on a specific threshold. After the motion detection kernel is applied, an `erosion` (morphology) filter is applied to reduce the noise. A `sobel` filter then highlights the edges. Finally, the edge information is combined with the original image in the `grayEdge2rgb` filter to highlight the motion in the image. The `gauss`, `erosion` and `sobel` kernels use 9×9 convolution filters. The RTCS identifies the convolutions and tiles them. In every iteration of the heartbeat loop, the application detects the motion between two different images.

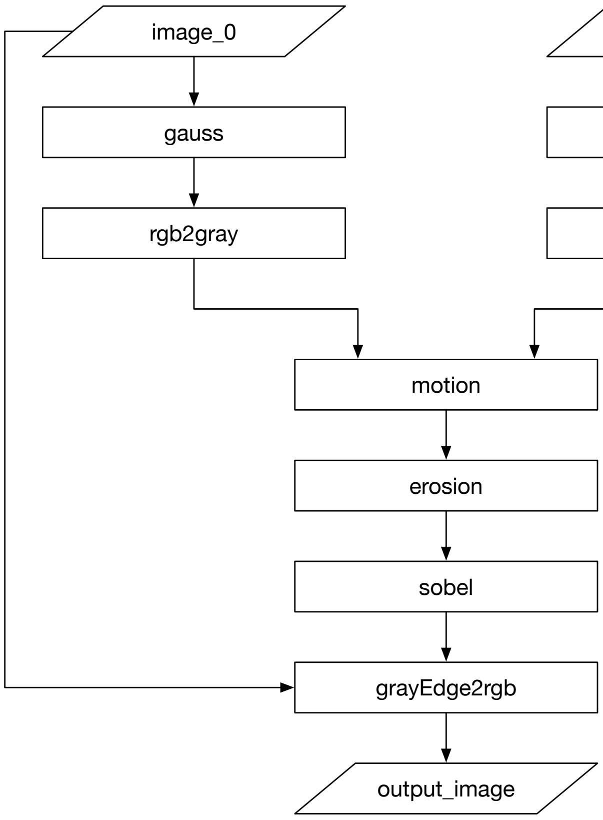


Figure 6.1: The `motion` application and its kernels.

6.2.8 Ray Tracing

Ray tracing is an image rendering technique commonly used in computer graphics. In computer graphics, a virtual camera is placed within a virtual environment and the resulting image is rendered from the point of view of this virtual camera. The direction in which this virtual camera is pointed is known as the viewing direction. The image plane is a plane that is perpendicular to this viewing direction and represents the image that the virtual camera will render. To calculate the color of a pixel in the image plane, the ray tracer constructs a ray that originates at the camera, passes through the center of the pixel and carries on into the virtual space. It then computes the closest intersection point between the objects in the scene and the ray. Finally, it determines the color of the object at the point of intersection and colors the corresponding pixel in the image plane. The `raytrace` application processes one ray in every heartbeat loop iteration and uses a simple implementation that computes the intersection points of a ray with all objects in the scene. All intersections can be computed in parallel and the kernel is parallelized by the RTCS.

6.2.9 SHA-256 Cryptographic Hash Function

SHA-256 is a cryptographic hash function with a digest length of 256-bits. The SHA-256 function takes a "message" and returns a fixed-size hash-value or "message digest". The hash function is not reversible and can only be used in one-direction. The `sha256` application represents the transform function of such a SHA-256 function. The function iterates over all the blocks and updates the message digest. All blocks can be processed in parallel and the kernel is parallelized by the RTCS, with a new message being processed in every heartbeat iteration.

6.2.10 Stereo Matching

The `stereo2D` application uses the `a2dOuter` function from the stereo matching test suite [Vaz14] that represent characteristic loops based on stereo matching algorithms. This function represents two loops where the inner loop has dependencies and cannot be parallelized. The RTCS is only able to parallelize the outer loop. Additionally, the loop is memory bound with very little computation. This example has been chosen to evaluate our approach in a scenario where parallelization is not ideal.

In the next section we look at the measuring method used to obtain the evaluation results.

6.3 Measuring Method

For our evaluation, the execution time is used as a performance metric. To measure the performance of the application and to obtain fine grained information about the different phases of a SAVE-Enabled application (SEA), SEAs are instrumented with lightweight timers. The RTCS is also similarly instrumented to measure its overheads and the time spent in its different phases. The SEAs are compiled into machine executable code by using the native `clang`/LLVM (v.3.8.0) compiler with optimization level -O3. The SEAs only implement sequential CPU code and serve as the baseline. The SEAs are also compiled into LLVM IR using the same `clang` front-end and are used as an input to the RTCS. To compare our approach against OpenACC, we extended the SEAs by adding pragmas to loops which instruct the compiler to generate parallelized code. We use the latest version of the PGI compiler (v.19.4-0) with optimization level -O3 to generate accelerated code for the supported multi-core CPU and the GPGPU devices. In our evaluation, we refer to these targets as `OpenACC-CPU` and `OpenACC-GPU` for the multi-core CPU and the GPGPU devices respectively.

Our evaluation platform supports three OpenCL-enabled accelerators: `mCPU`, `GPGPU` and the `PHI`. We evaluate the performance of the RTCS for each of these platforms.

Additionally, we also evaluate the performance of the `OpenACC-CPU` and `OpenACC-GPU`. For this evaluation, the offloading device is selected by the Orchestrator with the help of a debug parameter in its offloading decision module. In a production environment, the Orchestrator would base this decision on the application and system-level goals and policies in order to offload the computation to the best suited accelerator. However, as we want to evaluate all application variants on all accelerators, the offloading device is selected with the help of the Orchestrator debug parameter.

As the speedups depend on the input size, we do not select any single arbitrary input size, but use many different input variants to show the overall behavior including break-even-points to contrast the areas where offloading is un-/profitable. The `2mm`, `enhance`, `heat2D` as well as specific kernels from the `motion` are automatically tiled by the RTCS. We executed these applications by running each input data point for a fixed 16×16 tile size. In our approach, the work-group size for the generalized approach is not explicitly specified by the RTCS, but is automatically selected by the OpenCL runtime. For the tiling approach, the local work-group size is set to the tiling size.

All measurements are executed 25 times. The caching mechanism of the RTCS Server that reuses previously generated code, is turned off. To reduce the interference by the system OS, we use *tasksets* to pin the single-threaded baseline application to a single CPU. We also set the CPU governor to *performance* to prevent effects from dynamic frequency scaling. Even with these precautions, there were still some outliers. After the data was acquired, the outliers were automatically removed with the Interquartile Range (IQR) [51] method (factor $1.5 \cdot IQR$). Over all measurement points, the method marked and removed on average about 3 out of 25 runs. After eliminating the outliers, the average over all remaining runs was taken. To ensure that the code generated by the RTCS produces correct results, we use Google Test [28] to verify the outputs of our approach against the original CPU code. Since accelerators may induce small rounding errors for floating point computations, we tolerate values within a small threshold.

In this section, we looked at how we measure the benchmark applications, the different RTCS phases as well as how outliers are automatically removed. In the next section, we take an in-depth look at the overheads associated with the RTCS.

6.4 RTCS Overheads

In this section, we look at the following types of main overheads that are introduced by the Runtime and Just-in-Time Compilation System:

- Application launch overheads
- OpenCL code-generation and integration overheads
- OpenCL kernel compilation overheads
- Application-level OpenCL overheads

Figure 6.2 shows an overview of the different RTCS components and their overheads. The application launch overheads are the overheads introduced by the RTCS before it starts executing the application and are introduced by ② - ⑦. The code-generation overheads are introduced by the code-generation and integration modules of the RTCS Client and Server ⑪ - ⑭ and ⑯ - ⑱. Although, the OpenCL kernel compilation ⑮ can be considered to be a part of the code generation overheads, it has different characteristics for different OpenCL devices and to highlight them, we choose to present them separately. Overheads are also introduced into the application runtime by integrating the OpenCL environment into the application. These overheads are evaluated as application-level OpenCL overheads.

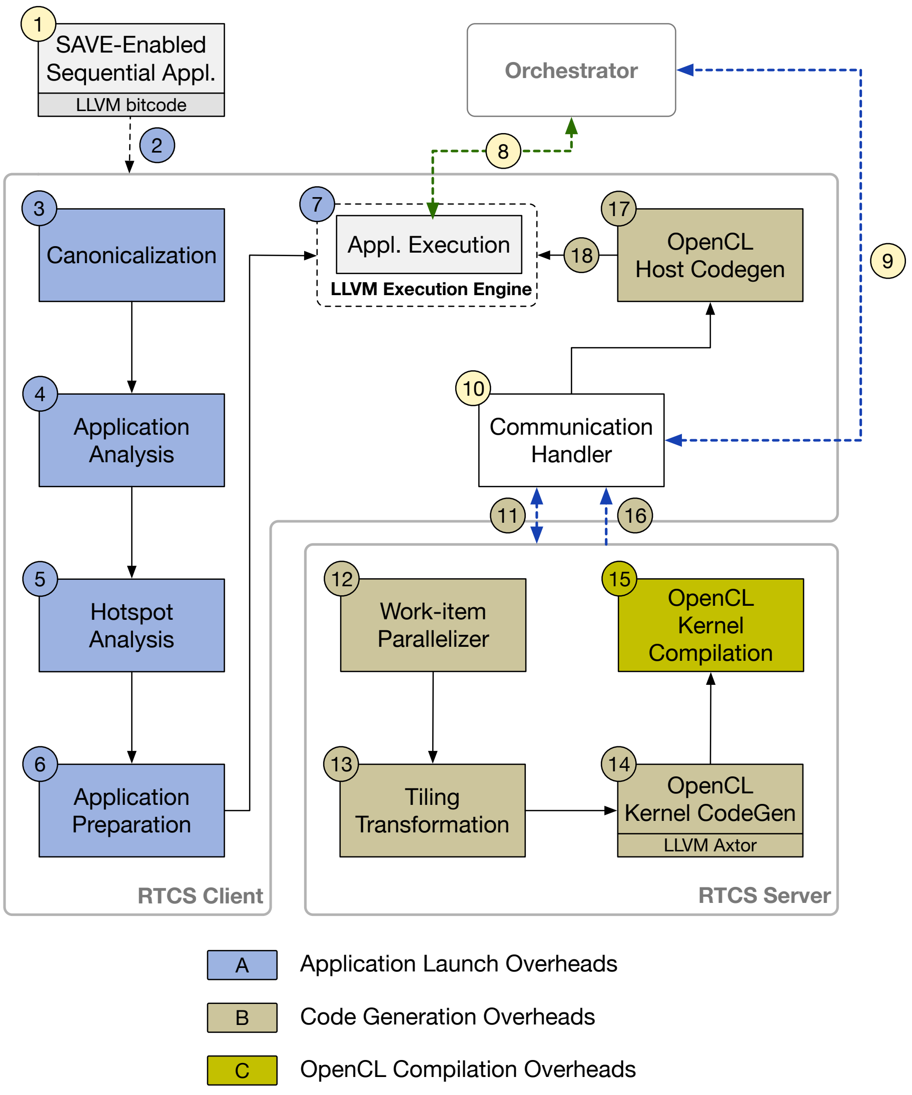


Figure 6.2: Overview of the RTCS overheads.

In addition to these overheads, the application also encounters the overhead of executing through the execution engine instead of native execution. The only difference between executing natively and via the execution engine is that the RTCS needs to preform JIT compilation. This is a one-time overhead and is considered as a part of the application launch overheads (Section 6.4.1). However, once JIT compiled, its effect on the actual performance of the application is negligible.

In the next sections, we will take a detailed look at the different RTCS overheads.

6.4.1 Application Launch Overheads

When a SEA is launched via the RTCS, the RTCS Client loads the application, analyses it and prepares it for acceleration before it starts executing the application. This overhead can be considered as the latency of the RTCS, i.e. the duration between the time the application is launched (via the RTCS) and the time when the application begins executing on the CPU. This overhead includes the time required for initializing the LLVM environment ②, canonicalizing the LLVM IR ③, application analysis ④, hotspot analysis ⑤, application preparation ⑥ and JIT compilation of the application via the execution engine ⑦. Some of these overheads are very small and have been grouped together into the following main categories:

(A_1) **LLVM initialization**: The LLVM environment is setup and the application is loaded into an in-memory LLVM module ②.

(A_2) **Canonicalization**: The LLVM IR is canonicalized to standardize it ③.

(A_3) **Application and Hotspot Analysis and preparation**: Orchestrator specific handles are detected in the application ④ and hotspots are analyzed for different kernels, external function calls, dependencies and possible data optimizations ⑤. The application is modified to support the OpenCL environment and accelerator hooks are added ⑥.

(A_4) **JIT Compilation**: The LLVM IR module is JIT compiled and the application starts executing ⑦.

Figure 6.3 represents these overheads for all the benchmark applications. The left axis of ordinates denotes the time in seconds while the right axis of ordinates denotes the size of the LLVM IR input file. The different benchmark applications along with the arithmetic and geometric means are represented on the axis of abscissas. We see that the LLVM initialization phase (A_1) is constant across all applications and is less than half a second. We can also see that the overhead for application analysis, hotspot analysis and application preparation (A_2) is very small. Most of the overheads are a result of the canonicalization (A_2) and the JIT compilation (A_4) phases. On an average, the canonicalization phase takes around 1 second while the JIT compilation phase requires around 3 seconds. From the figure, we can see that these phases are directly proportional to application size with larger applications requiring more time. The reason that JIT compilation dominates the overheads is because MC-JIT is used to JIT compile the application. One of the limitations of MC-JIT is that it has to compile the entire LLVM IR module before the execution engine can start executing the application. This overhead can be minimized by switching to ORC-JIT (available in newer LLVM releases) and using lazy compilation. These are one-time overheads and are incurred only when the application is launched, but still need to be considered when launching applications via the RTCS.

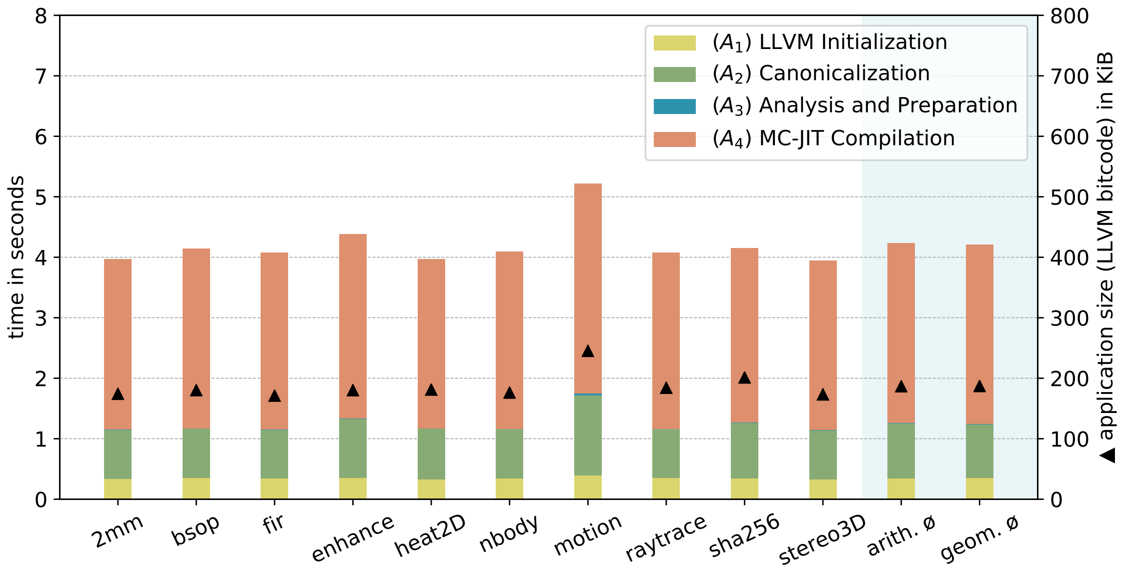


Figure 6.3: The initialization, analysis and application preparation overheads of the RTCS.

6.4.2 OpenCL Code Generation and Integration Overheads

The code generation overheads are associated with the OpenCL code generation and integration steps of the RTCS. They include communication and LLVM IR exporting overheads (11), work-item parallelization (12), tiling transformations (13), OpenCL kernel code generation (14), transfer of the OpenCL kernel to the RTCS Client (16) as well as OpenCL host code generation (17) and JIT compiling the registration function and integration of the new accelerator implementation into the application (18). The RTCS concurrently generates accelerated code while the application is executing. The RTCS Client runs on a different thread (core) as the application, while the RTCS Server runs as a separate application that can be executed on a separate node. These overheads do not directly affect the application execution time, but since they require CPU computation time, they might indirectly influence the performance of the applications running on an already highly utilized system. The overheads of some of the code generation steps are very small and for the sake of clarity, we have grouped them into the following categories:

(B_1) **Export LLVM IR**: This includes the communication overheads (11) required to request the codegen and transfer the HDS between the RTCS Client and Server along with the time required by the Client to export the hotspot and kernels into LLVM IR and send it to the Server.

(B_2) **OpenCL kernel code generation**: The time required by the work-item parallelization (12), tiling transformations (13) and OpenCL kernel code generation (14).

(B_3) **OpenCL host code generation**: The transfer of the OpenCL kernel to the Client (16) as well as the time required to generate corresponding OpenCL host code (17).

(B_4) **Application integration**: Updating and JIT compiling the registration function (18).

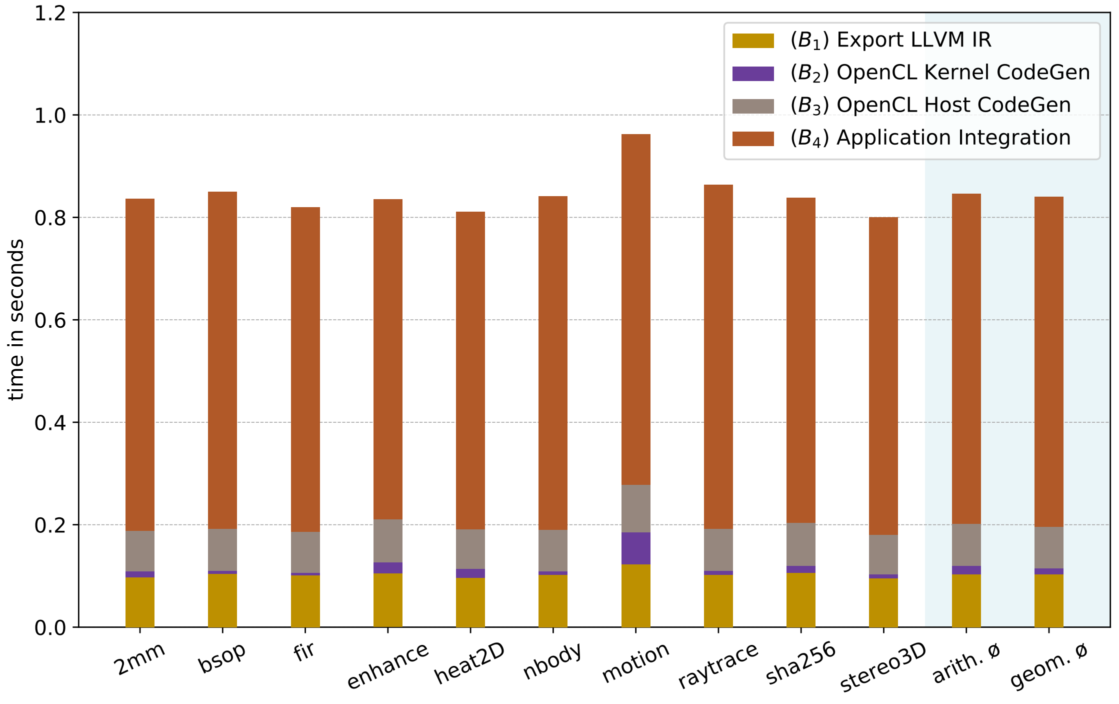


Figure 6.4: The code generation overheads of the RTCS.

Figure 6.4 shows the code generation overheads for all benchmark applications. The axis of ordinates denotes the time in seconds while the different benchmark applications along with the arithmetic and geometric means are represented on the axis of abscissas. We see that the overheads are quite similar for all the applications. We see that `motion` which has six kernels requires more time to export the hotspot and kernel LLVM IR (B_1). It also takes more time to generate the OpenCL kernel code (B_2). Applications like

`2mm`, `enhance` and `heat2D` that are tiled, have a marginally higher OpenCL kernel code generation time (B_2) when compared to the rest. The OpenCL host code generation (B_3) takes around 82 milliseconds and is nearly constant across all benchmark applications. The application integration phase (B_4) is responsible for the largest code generation overhead and on average takes around 645 milliseconds. The application integration phase creates a new lambda function and inserts code into the registration function and also JIT compiles this registration function. The major contributor to this application integration overhead is the *clang* compilation step required to generate LLVM IR from C++ code. The code generation overheads across all the applications is quite small (less than 1 second) and do not directly affect the execution time of the application. By making use of the code caching mechanism of the Server, overheads introduced by phases (B_1) and (B_2) can be reduced.

6.4.3 OpenCL Kernel Compilation Overheads

To execute the OpenCL C kernel code on the device, it needs to be compiled into device specific binaries (15). This is performed by the RTCS Server and in this section, we focus on the overheads associated with the OpenCL kernel compilation for different OpenCL devices.

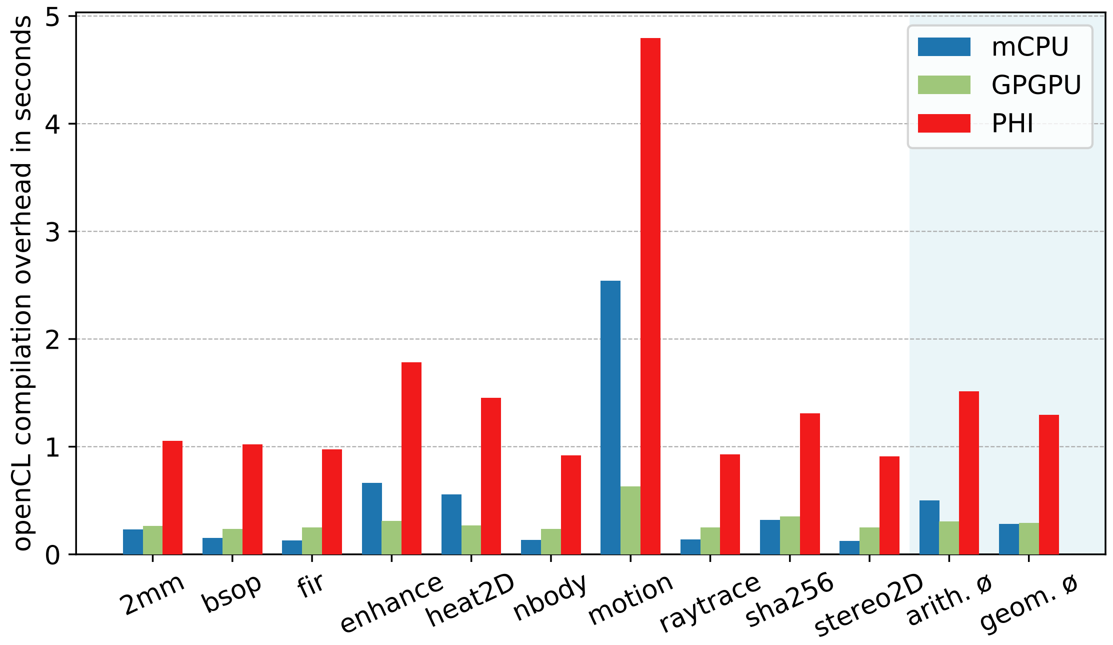


Figure 6.5: OpenCL compilation overheads for different OpenCL devices.

Figure 6.5 shows the OpenCL compilation time in seconds for the application kernels across the different OpenCL devices of the heterogeneous platform. The axis of ordinates denotes the OpenCL time required to compile the OpenCL C kernel code in seconds, while the different OpenCL devices for each benchmark application are grouped together on the axis of abscissas. The arithmetic and geometric means of each individual device is also represented on the axis of abscissas. We can see that in most cases, the compilation time for the `mCPU` is the fastest, followed by the `GPGPU` and then the `PHI`. All applications have only one OpenCL kernel except for `motion`, which has six kernels. Looking at `motion` in Figure 6.5, we can see that the compilation of six kernels contributes to a large increase in the compilation time for the `mCPU` and `PHI` devices. Looking at the `GPGPU` compilation time across all applications, we see that it is relatively constant when compared to the behavior of the `mCPU` and `PHI`. The OpenCL compilation for the Nvidia platform is more efficient than the Intel platform and we only see a marginal increase in the `GPGPU` compilation time for the more complex `motion` application.

We can also see that on average, OpenCL compilation for the `PHI` is more than 3$\times$ slower than the `mCPU` and the `GPGPU`. These overheads, however, do not directly contribute to the overall execution time of the application. Similar to the B_1 and B_2 overheads presented in Section 6.4.2, the RTCS Server's code caching mechanism can be used to reduce these OpenCL compilation overheads.

6.4.4 Application-level OpenCL Overheads

In Section 5.8 we saw that code to create OpenCL device handles and command queues for all the missing OpenCL devices was added to the application. Additionally, once a new accelerated implementation is integrated into the application, the application loads the OpenCL kernel binary and registers the implementation with the Orchestrator. These steps are performed within the execution context of the application and directly contribute to the execution time of the application. In our evaluation, it was found that the overheads attributed to initializing the OpenCL devices and registering the implementation with the Orchestrator was negligible. Hence, in this section we focus on the overheads associated with loading the OpenCL kernel binary into the program.

Figure 6.6 shows the OpenCL binary integration time in seconds for the application kernels across the different OpenCL devices. The axis of ordinates denotes the OpenCL binary integration time in seconds while the different OpenCL devices for each benchmark application are grouped together on the axis of abscissas. The arithmetic and geometric means of each individual device is also represented on the axis of abscissas. Looking at the figure, we can see that the amount of time required by each device to load a pre-compiled OpenCL kernel is nearly constant across all benchmark applications. In most cases, the `mCPU` is marginally faster than the `GPGPU`. However, the time required by the `PHI` to load an OpenCL kernel binary is more than 4$\times$ slower than the `mCPU` and `GPGPU`. Although these overheads are relatively small and less than 1 second (smaller for the `mCPU` and `GPGPU`), they contribute to the overall execution time of the application and need to be amortized. They are however one-time overheads and once the OpenCL code for a specific accelerator has been loaded, the Orchestrator can offload computation to the accelerator multiple times during the application's lifetime.

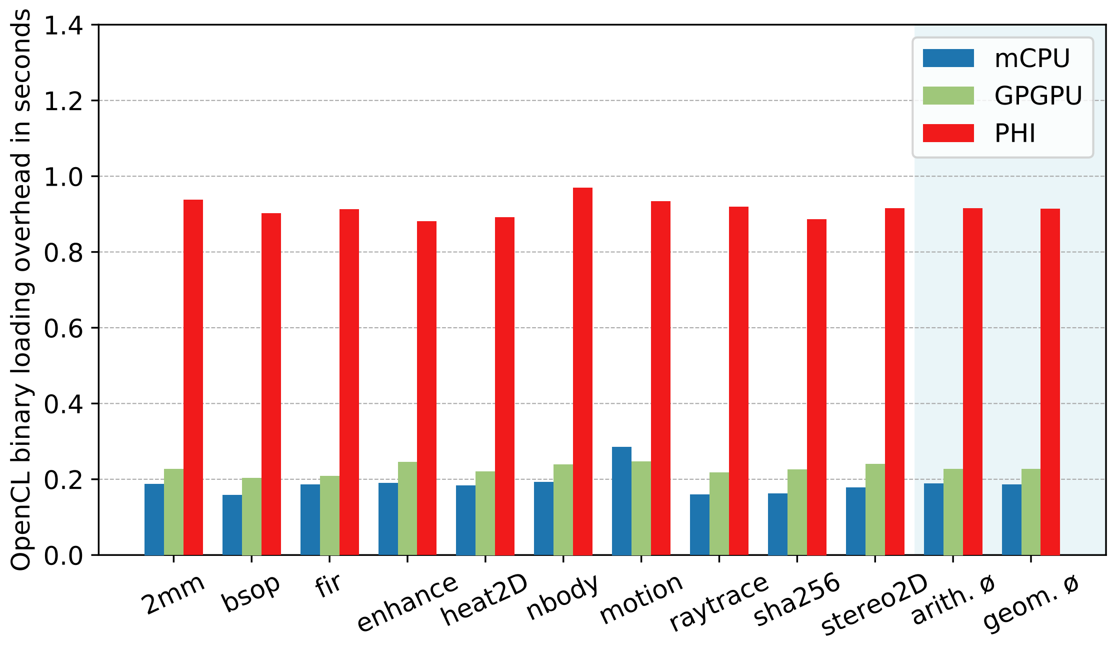


Figure 6.6: Loading overhead of the OpenCL binary for different OpenCL devices.

In the previous sections, we looked at the different types of direct and indirect RTCS overheads. In the next section, we evaluate the performance of the RTCS.

6.5 Performance Evaluation

In this section, we present the main outcome of this work, by evaluating the improvement in application performance as a direct result of the RTCS. To determine this, the sequential baseline is compared to the OpenCL-enabled parallel implementation. Even though the focus of this thesis is on the overall method and tool chain, the aim is to improve the performance of sequential applications. We evaluate the performance at the *kernel-level* as well as at the *hotspot-level* for all benchmark applications. Figure 6.7 shows what is measured at the kernel-level and hotspot-level. The kernel-level evaluation focuses on the evaluation of the pure kernel speedups while the hotspot-level evaluation looks at the performance of the entire hotspot and also includes the data transfer overheads. The runtime of the entire applications depends on the number of heartbeat iterations which can be arbitrarily selected to influence the evaluation results. Hence, we do not perform any application-level performance evaluation, but instead only focus on the kernel and hotspot-level analysis.

In Figures 6.8, 6.9 and 6.10, we take an in-depth look at the kernel and hotspot-level speedups and plot the input sizes on the axis of abscissas. We present the `fir`, `heat2D` and `2mm` applications that form a representative subset of the benchmark applications.

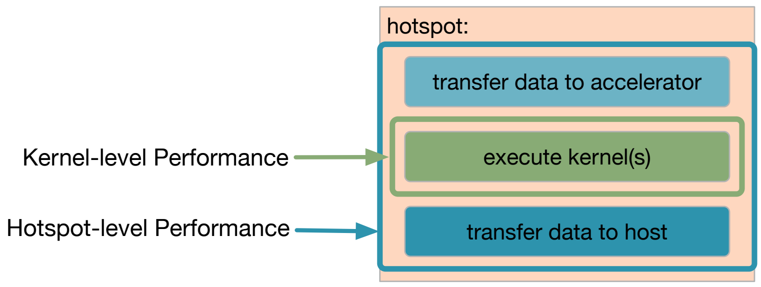


Figure 6.7: Measurement of the kernel-level and hotspot-level performance.

The figures for the remaining benchmark applications are made available in the appendix. The hotspot numbers in the bottom plots (b), include kernel execution and data transfer overheads. We also compare our approach against a pragma-based OpenACC production compiler. The `heat2d` and `2mm` applications benefit from the RTCS's tiling optimizations.

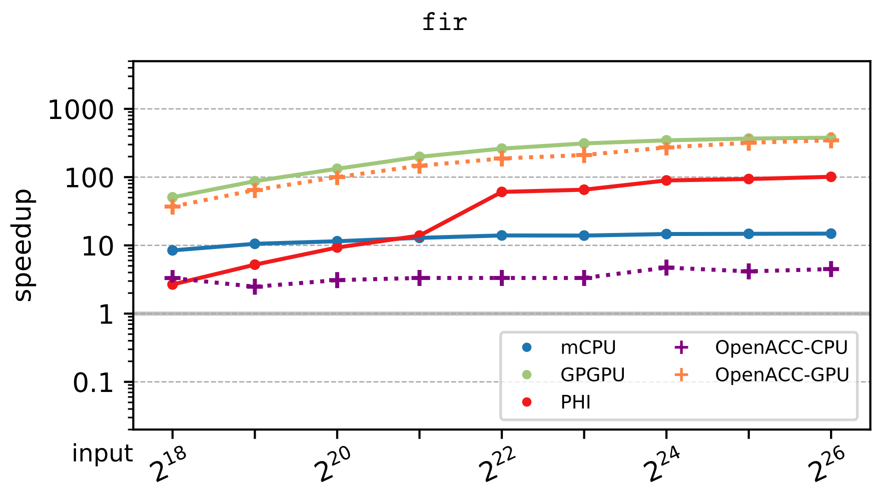
6.5.1 Kernel-level Evaluation

We first evaluate the kernel-level performance where we look at the pure kernel speedup and compare the baseline kernel against our approach for the different OpenCL accelerators (`mCPU`, `GPGPU` and `PHI`). Looking at the kernel speedup of our approach for all three benchmarks (Figures 6.8a, 6.9a and 6.10a), it can be seen that for larger input sizes, the `GPGPU` performs best and delivers the highest speedup. The `PHI` outperforms the `mCPU`, but is considerably slower than the `GPGPU`. However, this trend does not hold for smaller input sizes where the `mCPU` performs better than the `PHI`. This behavior can be attributed to device specific initialization overheads where the initial low-level efforts required to start executing multiple parallel threads on the device are not completely amortized for smaller input sizes.

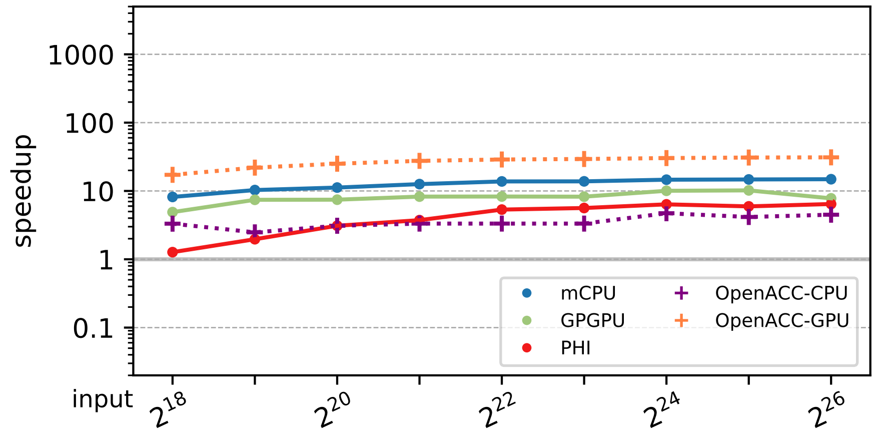
Looking at Figures 6.9a and 6.10a it can be observed that for `heat2D` and `2mm`, this overhead is so large that we initially see a slowdown on the `PHI` for smaller input sizes. However, as the input sizes of `heat2D` and `2mm` increase, these initialization overheads are amortized and the performance on the `PHI` improves allowing it to eventually outperform the `mCPU`.

In Figures 6.8a and 6.9a, one can see that as the input sizes of `fir` and `heat2D` increases, the kernel speedup begins to plateau. This indicates the saturation point or the maximum speedup that can be achieved by the given OpenCL kernel and device combination. This maximum kernel speedup is also indicative of the accelerator type, where the more powerful `GPGPU` performs best.

Comparing our approach against a pragma-guided OpenACC approach in Figures 6.8a, 6.9a and 6.10a, we can see that the RTCS kernel speedups are comparable and, in most cases, better than OpenACC. Looking at the speedup for `fir` in Figure 6.8a, we can see that the speedup for the `GPGPU` and `OpenACC-GPU` are exactly the same, while in the case of `2mm` (Figure 6.10a) the `GPGPU` performs slightly better than the `OpenACC-GPU`. However,



(a) Kernel.



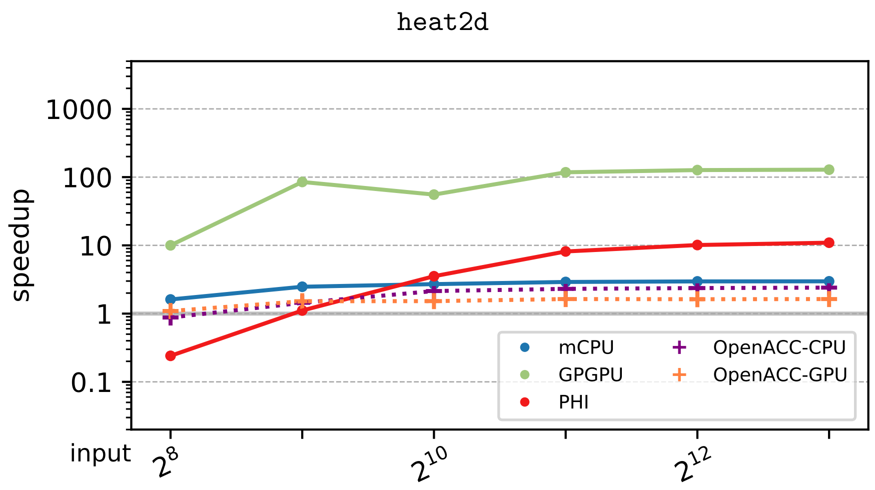
(b) Hotspot including data transfer.

Figure 6.8: `fir`: Kernel and hotspot-level speedup for different input sizes. Plot (a) shows the kernel-level speedup, while plot (b) shows the hotspot-level speedup including data transfer overheads.

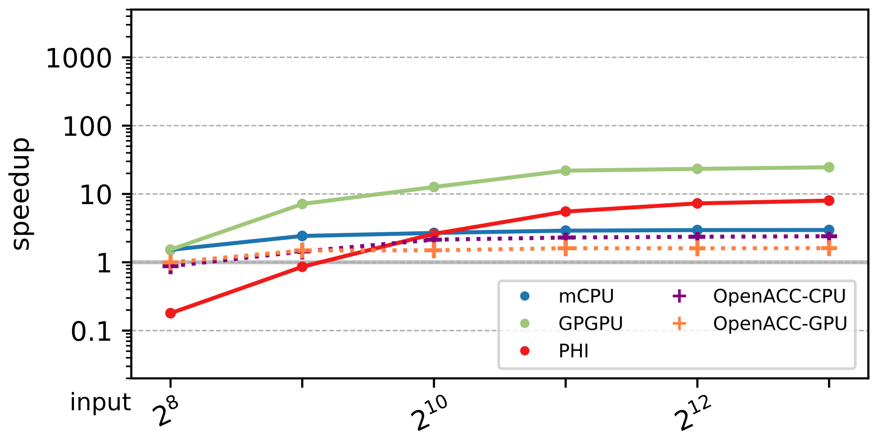
`heat2D` (Figure 6.9a) benefits from the tiling optimizations performed by the RTCS and the `GPGPU` performs substantially better than the `OpenACC-GPU`.

6.5.2 Hotspot-level Evaluation

Although kernel speedups play an important role in determining the performance of our RTCS code generation approach, we also need to evaluate the overall performance in the real world, where data needs to be transferred to the accelerator. Figures 6.8b, 6.9b and 6.10b show the hotspot-level performance improvement for the corresponding `fir`,



(a) Kernel.



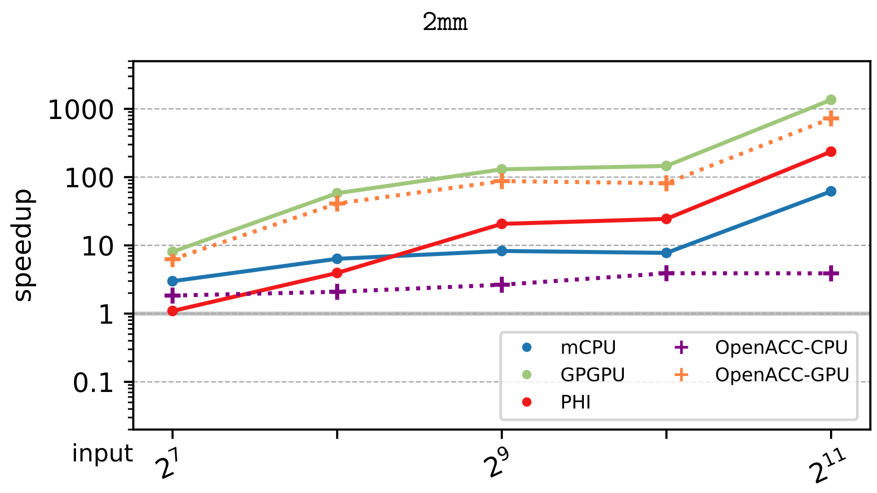
(b) Hotspot including data transfer.

Figure 6.9: `heat2D`: Kernel and hotspot-level speedup for different input sizes. Plot (a) shows the kernel-level speedup, while plot (b) shows the hotspot-level speedup including data transfer overheads.

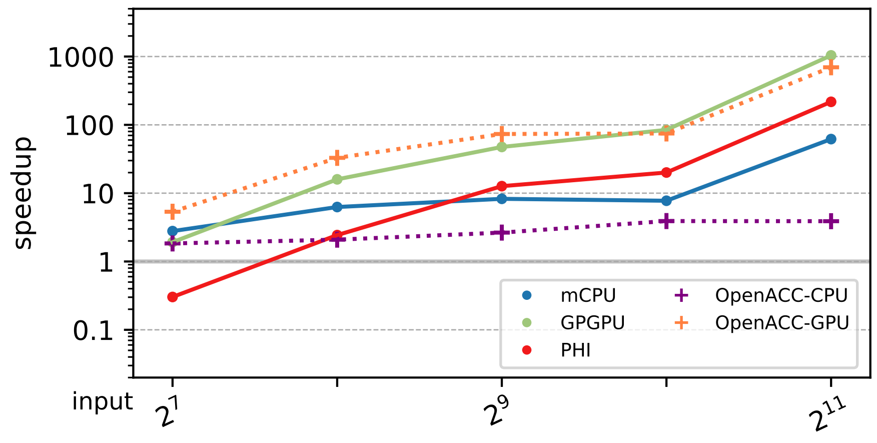
`heat2D` and `2mm` applications over increasing input sizes.

This hotspot-level speedup also includes the data transfer overheads required to transfer data from the host to the OpenCL device and back as well as the overhead of issuing commands to create OpenCL buffers on the device, parametrize and enqueue the kernel and finally free the device buffers. Although the hotspot-level speedups are computed using all these overheads, we only focus on the data transfer overheads as the other overheads are negligible.

Looking at Figures 6.8, 6.9 and 6.10 and comparing the kernel-level and hotspot-



(a) Kernel.



(b) Hotspot including data transfer.

Figure 6.10: `2mm`: Kernel and hotspot-level speedup for different input sizes. Plot (a) shows the kernel-level speedup, while plot (b) shows the hotspot-level speedup including data transfer overheads.

level speedups for `fir`, `heat2D` and `2mm`, we can see that in general, the hotspot-level speedups are lower than the corresponding kernel-level speedups for the `GPGPU`, `PHI` and `OpenACC-GPU`. This is because of the data transfer overheads that need to be amortized. In the case of the `mCPU` and the `OpenACC-CPU`, the host and the accelerators share the same (host) memory. In this scenario, the RTCS as well as OpenACC are able to skip the data transfer steps and achieve identical kernel-level and hotspot-level speedups.

On a closer comparison of the hotspot-level slowdown with respect to the kernel-level performance, we see that in the case of `heat2D` (Figure 6.9) the `GPGPU` is affected the most but the drop in performance is nearly constant across all input sizes, while there is no significant slowdown for the PHI and the OpenACC-GPU. Looking at the GPGPU for the computationally intensive 2mm application (Figure 6.10) we see that we have larger slowdowns for smaller input sizes, but as the computation time dominates the data transfers for larger input sizes, we do not see a considerable difference between the kernel-level and hotspot-level performance. On the other hand, the data hungry fir application (Figure 6.8) requires more time to transfer larger inputs to the accelerator which in turn needs to be amortized. Looking at the figure, we see that as the input size increases, the overall performance at the hotspot-level remains relatively constant. This is because, although the larger input sizes result in more computation, they are offset by the larger data transfer times that are required to be amortized. Figure 6.11 shows the behaviour of the hotspot-level performance of the fir application in relation to its kernel-level performance for different input sizes. As the application input size increases, we see a decrease in the relative performance of the GPGPU, the PHI as well as the OpenACC-GPU. These data transfer overheads are so large, that at the hotspot-level, the mCPU performs better than the GPGPU and PHI (see Figure 6.8b) even though the GPGPU and PHI outperform it at the kernel-level.

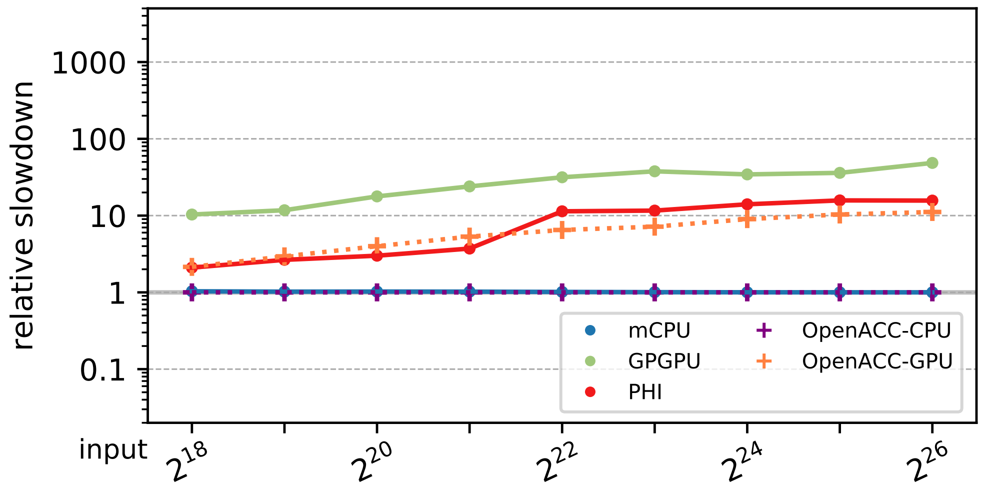


Figure 6.11: fir : The slowdown at the hotspot-level when compared to the kernel-level performance.

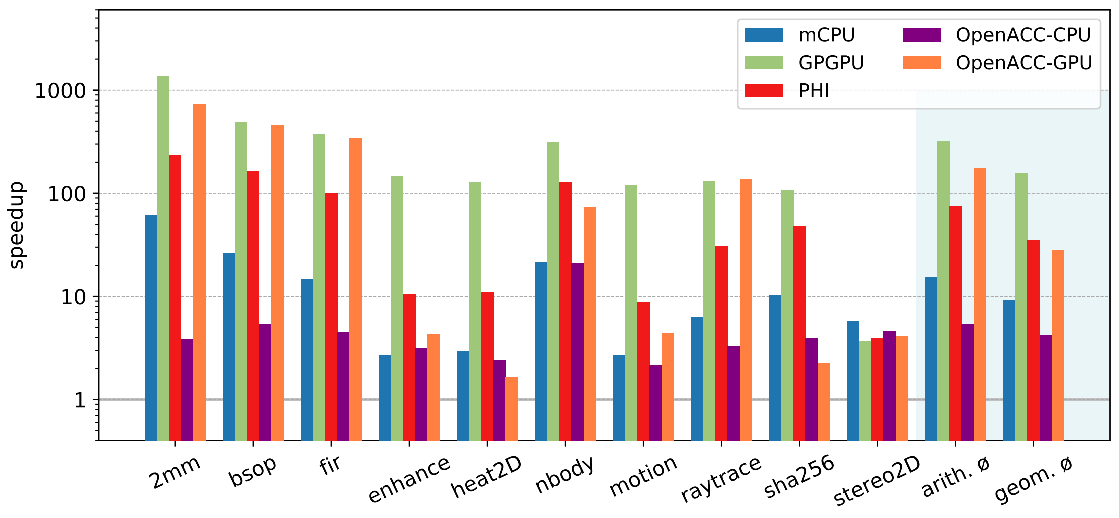
Overall, we have seen a drop in performance at the hotspot-level when compared to the pure kernel-level. This affects the break-even point, i.e. the point where data overheads can be amortized and offloading to the accelerator starts to get profitable (speedup > 1). For heat2D and 2mm, in Figures 6.9 and 6.10 we see that the PHI reaches its hotspot-level break-even point at a slightly higher input size as compared to its kernel-level breaking-point. Although we see a drop in relative performance at the hotspot-level for the different kernel/accelerator combinations, we still achieve an overall performance improvement as compared to the baseline, allowing us to profit by offloading computation to heterogeneous accelerators.

6.5.3 Performance across all Benchmark Applications

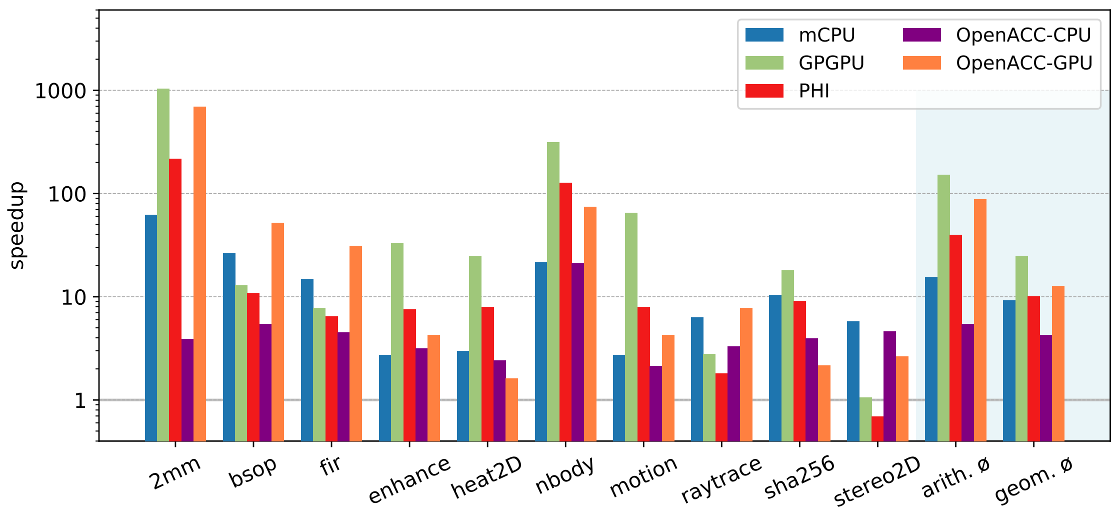
In Sections 6.5.1 and 6.5.2 we took an in-depth look at the kernel-level and hotspot-level speedups for different input parameters. Looking at the application's performance over increasing input sizes we saw that the speedup slowly increases and saturates after reaching a specific input size (see Figure 6.8). These saturation points of the kernel-device combination are used in Figure 6.12 to summarize the kernel-level and hotspot-level speedups for all baseline applications. The time complexity of the sequential baseline `2mm` application that multiplies two $n \times n$ dense matrices is given by $\mathcal{O}(n^3)$. This is cubic in nature and results in extremely long evaluation times for larger input sizes. We were not able to determine the saturation point for `2mm` (Figure 6.10) and use the speedup achieved for the largest evaluated input size instead.

Looking at the overall performance trend of the RTCS for each application in Figure 6.12, we see the same trends that were observed in Sections 6.5.1 and 6.5.2, where the `GPGPU` performs the best and is followed by the `PHI` and then the `mCPU`. We see that computationally intensive applications like `2mm`, `nbody` and `motion` are able to effectively amortize data transfer overheads resulting in similar kernel-level and hotspot-level speedups. We also see an overall hotspot-level speedup across all devices for all applications with the exception of `stereo2D`. The `stereo2D` application is memory bound with very little computation and although we see kernel speedups, the data transfer overheads are too large to be amortized, resulting in no hotspot-level speedup for the `GPGPU` and even results in a slowdown for the `PHI`.

Looking at Figure 6.12 and comparing the kernel-level performance of `stereo2D` for the `GPGPU` and the `OpenACC-GPU`, we see that they are nearly equal. However, at the hotspot-level, we see a more significant decrease in the `GPGPU` speedup compared to the `OpenACC-GPU`. Looking at `bsop`, `fir`, `heat2D`, `raytrace` and `sha256` we find a similar pattern. On further investigation, we found that the data transfers to/from the GPGPU are 7× faster with OpenACC as compared to OpenCL. This stems from the differences in how the OpenCL runtime handles data transfers as compared to how the PGI compiler internally uses CUDA calls to transfer data. In the case of `bsop` and `fir` this results in the `mCPU` being faster than the `GPGPU` and `PHI` for larger input sizes. However, despite this we still see hotspot-level speedups across all accelerators for 9 out of the 10 benchmark applications with our approach performing better than OpenACC on average. Across all applications, we see average hotspot-level speedups of 15×, 150× and 40× for the `mCPU`, `GPGPU` and `PHI` respectively. This average is however dominated by the speedups from `2mm` and `nbody`. Looking at the geometric mean, we see hotspot-level speedups of 9×, 25× and 10× for the `mCPU`, `GPGPU` and `PHI` respectively, with the `mCPU` benefiting from no data transfer overheads. By being capable of generating code targeting different heterogeneous accelerators, the RTCS is able to offer a considerable performance improvement over sequential applications.



(a) Kernel.



(b) Hotspot including data transfer.

Figure 6.12: Kernel and hotspot-level speedups for all benchmark applications: Figure (a) shows the performance improvement of the kernel, while Figure (b) shows the performance improvement of the entire hotspot including data transfer overhead.

In the previous sections, we evaluated the performance of the RTCS at the kernel-level as well as the hotspot-level and compared it against a pragma-based OpenACC production compiler. For this evaluation, the RTCS used its tiling mechanism to improve the performance of some of the benchmark applications. Additionally, the hotspot-level evaluation also benefits from the data transfer optimization performed by the RTCS. In the next sections, we evaluate the performance of our tiling approach as well as the data transfer optimizations.

6.6 Tiling Performance

In Section 5.11 we saw how the RTCS is able to improve data reuse by tilling the OpenCL kernels using two different tiling approaches. In this section, we evaluate the improvement in kernel-level performance as a result of our tiling approach. Out of the 10 benchmark applications, `2mm`, `enhance`, `heat2D` and parts of `motion` are automatically tiled by the RTCS. `motion` contains multiple kernels (see Section 6.2) out of which only `gauss`, `erosion` and `sobel` can be tiled. The `2mm` application is tiled using the matrix multiplication approach, while the remaining kernels are detected as convolutions. As previously mentioned in the measurement method (Section 6.3), we use a fixed tile size of 16×16 in our evaluation. The performance improvement is represented as a speedup fraction obtained by comparing the kernel execution time of our tiled approach to the kernel execution time of our generalized approach (that only makes use of global memory). Figures 6.13, 6.14, 6.15 and 6.16 show the tiling speedup over different input sizes for `2mm`, `enhance`, `heat2D` and `motion` respectively.

The RTCS is able to detect a matrix multiplication pattern in the computationally intensive `2mm` application and is able to apply tiling optimizations to the OpenCL kernel. Looking at Figure 6.13, we can see that for the smallest input size, tiling does not yield any significant performance improvements with only the `PHI` showing marginal performance gains. However, as the input size increases, we see a considerable improvement in the performance of the `GPGPU`. In most cases however, we do not see any significant difference in the performance of the `mCPU` or `PHI` except for the largest input size, where we see a $6\times$ speedup for the `mCPU`. The speedup achieved by our matrix multiplication tiling approach for the `GPGPU` is around $35\times$ for the largest input size and around $14\times$ on average, which is a considerable improvement over a non-tiled approach.

The `enhance` application is an image sharpening application and works on images with three components (RGB) per pixel and uses a 9×9 convolution filter. The RTCS is able to identify the convolution and apply tiling optimizations to the OpenCL kernel. Looking at Figure 6.14, one can see that similar to `2mm`, tiling does not improve the performance of the `mCPU` and `PHI`. We however see an increase in the performance of the `GPGPU` as the input size increases and achieves an average tiling speedup of around $10\times$. `heat2D` application is also based on a convolution filter, but unlike `enhance` uses only one component and internally makes use of a smaller 5×5 convolution filter. This results in different data reuse and computation patterns which affect the tiling speedup. Looking at Figure 6.15 we initially see small slowdowns for the `mCPU` and `PHI` for smaller input sizes. However,

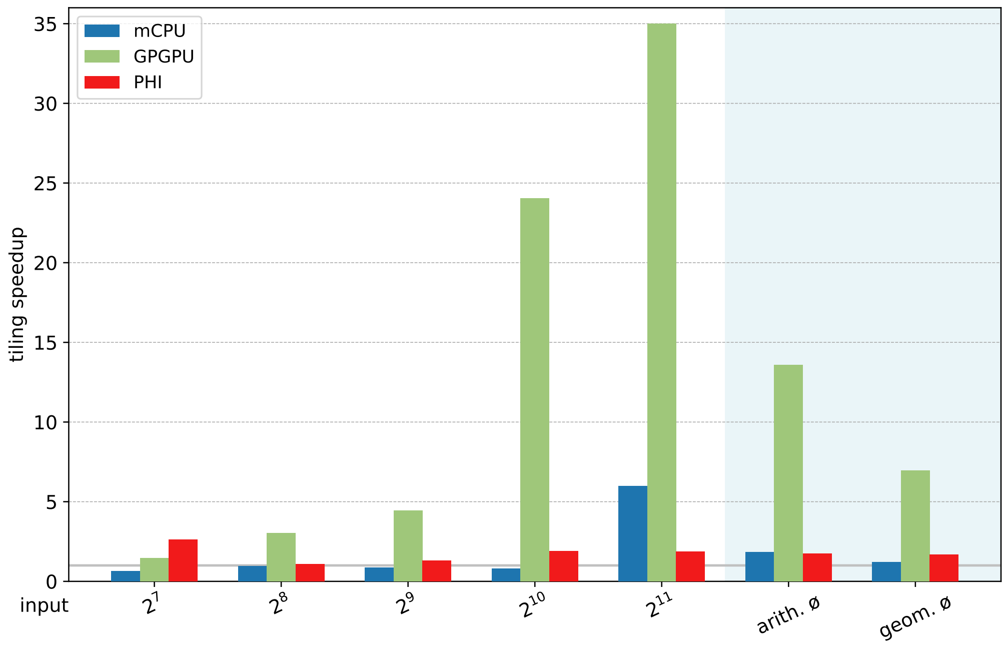


Figure 6.13: `2mm` : Kernel speedup when using tiling and local memory.

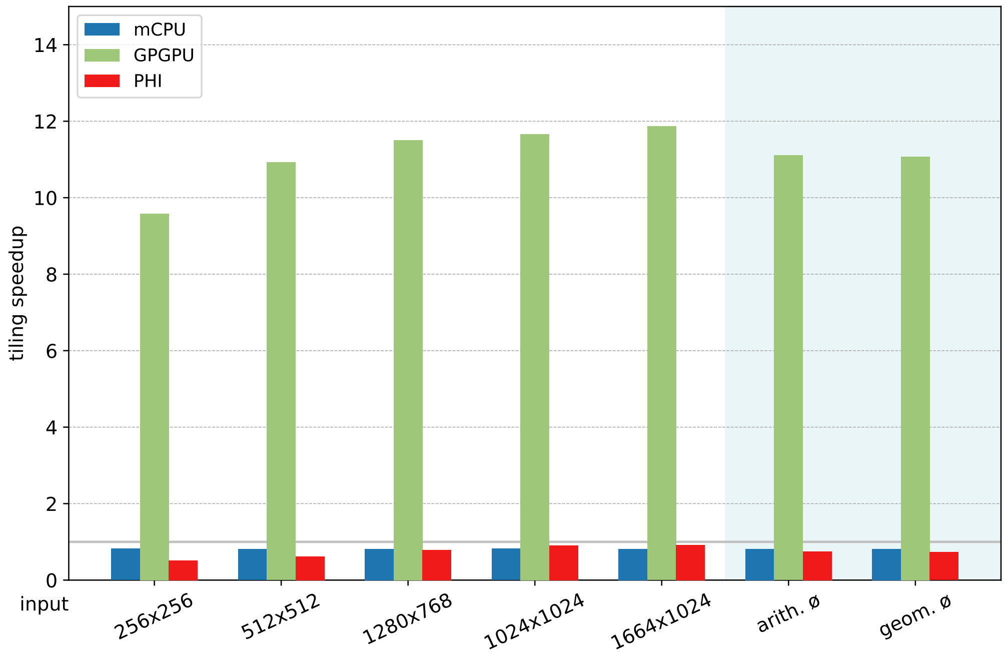


Figure 6.14: `enhance` : Kernel speedup when using tiling and local memory.

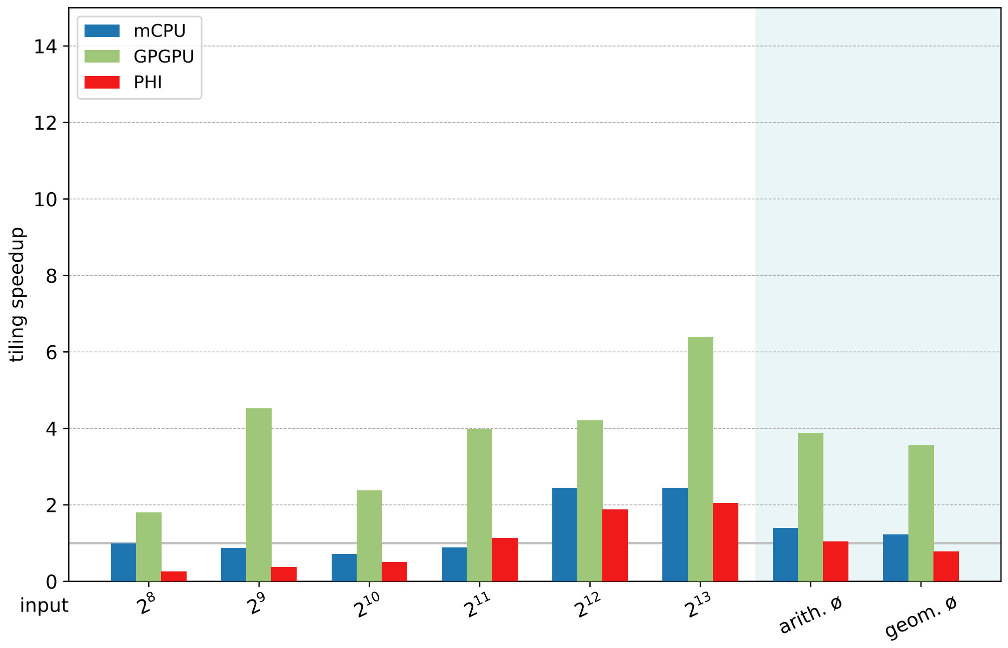


Figure 6.15: `heat2D` : Kernel speedup when using tiling and local memory.

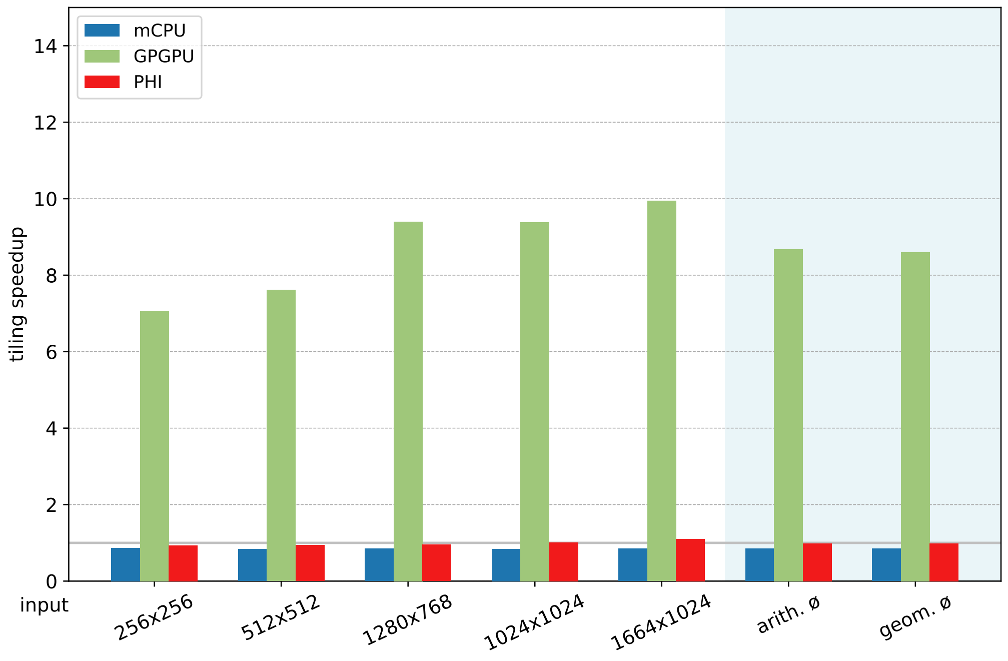


Figure 6.16: `motion` : Kernel speedup when using tiling and local memory.

as the input size increases, we see an increase in the tiling performance and eventually, we see a speedup for the `mCPU` and `PHI` devices. The `GPGPU` also exhibits a general trend of improved tiling performance as the input size increases and achieves an average tiling speedup of 4×.

The `motion` application contains six different kernels (see Section 6.2). Out of these kernels, `gauss`, `erosion` and `sobel` are based on convolution filters and can be tiled by the RTCS. Similar to `enhance`, these kernels work on images with three components (RGB) per pixel and uses a 9×9 convolution filter. Figure 6.16 shows the overall speedup across all the kernels (tiles and non-tiled) in `motion`. Similar to `enhance` (Figure 6.14), we see that tiling does not improve the performance of the `mCPU` and `PHI`. We also see that the `GPGPU` follows a similar trend as `enhance`, however, with a slightly lower speedup. This difference can be attributed to additional kernels in `motion` that are not convolution filters and cannot benefit from the tiling optimizations of the RTCS.

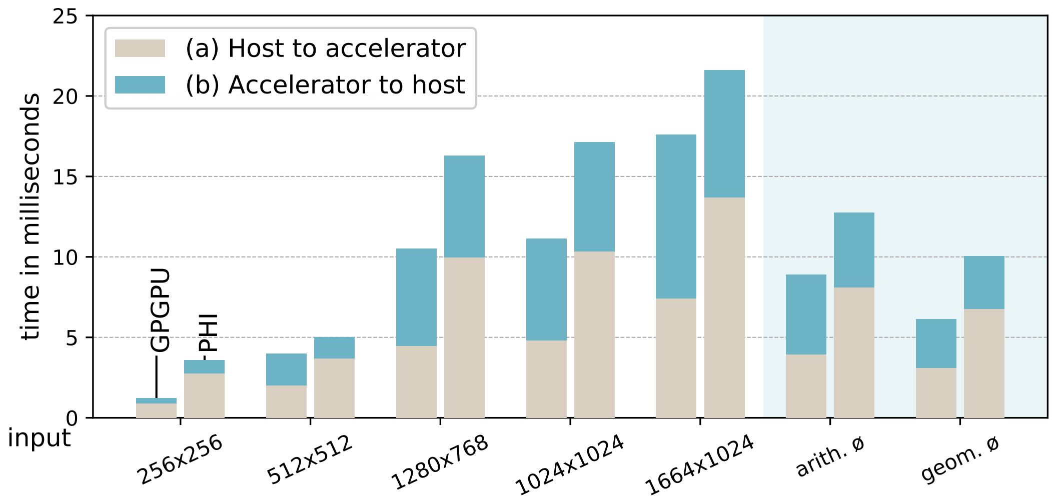
Looking at the overall tiling results, we see that the amount of data reuse plays a key factor in the tiling speedups, where matrix multiplication performs substantially better than convolutions. We also see that as the input size increases, so does the tiling speedup. In most cases and for smaller input sizes, we do not see any tiling speedups for the `mCPU` and `PHI`. However, we see that our tiling approach is most effective for the `GPGPU` and gives us a considerable performance improvement over the generalized approach.

6.7 Data Transfer Optimizations

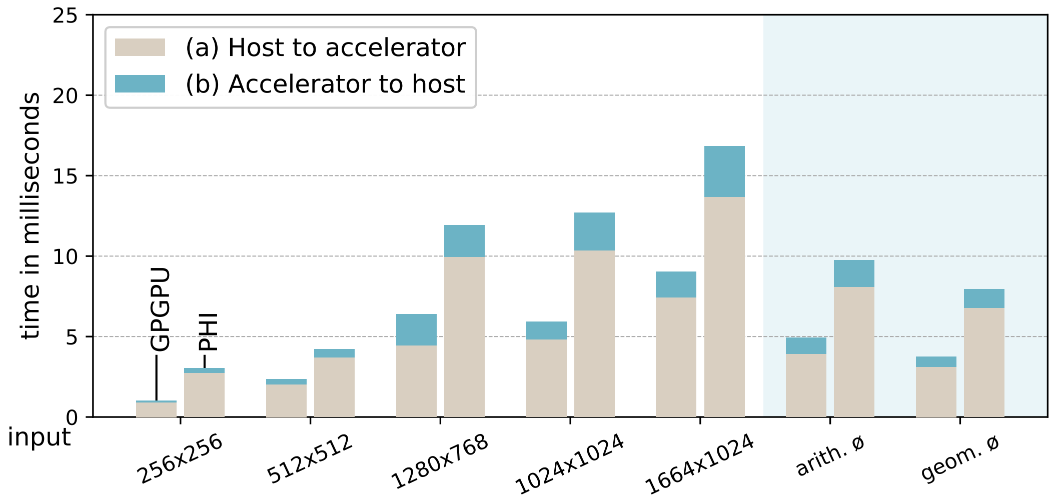
In the hotspot-level evaluation section (Section 6.5.2), we saw that data transfer overheads have a major effect on the overall hotspot speedup. In Section 5.7.3 we looked at how the RTCS performs data transfer optimizations by detecting which OpenCL buffers are modified by the kernel and only transferring the modified OpenCL buffers back to the host. In this section we evaluate the effectiveness of this data transfer optimization. To evaluate the data transfer optimization, we consider the `motion` application. We only evaluate the `GPGPU` and `PHI` as the RTCS does not transfer data to the `mCPU`, but uses a pointer to the host memory instead.

Figure 6.17 shows the OpenCL data transfer overheads for the `motion` application. The axis of ordinates denotes the time in milliseconds while the different input sizes along with the arithmetic and geometric means are represented on the axis of abscissas. The grey bars represent the amount of time required to transfer the data from the host to the device while the sky-blue bars represent the time required to transfer data from the device to the host. Looking at Figure 6.17a, we can see that the data transfer times for the `GPGPU` are faster than the `PHI` for all input sizes. This is because, when compared to the `GPGPU`, the `PHI` takes longer to transfer the same amount of data from the host to the device. We also see that the amount of time required for the transfers increases with the increase in input size.

Figure 6.17a shows the data transfer overheads without any RTCS optimizations while Figure 6.17b shows the same overheads but with RTCS data transfer optimization. Comparing the sky-blue bars between Figure 6.17a and 6.17b we can clearly see the benefits of



(a) Unoptimized OpenCL data transfer overheads.



(b) Optimized OpenCL data transfer overheads.

Figure 6.17: Unoptimized and Optimized OpenCL data transfer overheads over different input sizes for `motion`.

our data transfer optimization and that data transfer optimizations considerably reduce the overall data transfer times. From Figure 6.17b we can also see that since only the modified buffers are transferred back to the host, the time required to transfer data back to the host is considerably smaller than the time required to transfer data from the host to the accelerator.

In the `motion` example, this results in an accelerator to host data transfer saving of around 80% for the `GPGPU` and 64% for the `PHI`. Looking at the complete data transfer overheads, we gain a saving of around 44% for the `GPGPU` and 23% for the `PHI`. Data transfer overheads affect the overall speedup of the hotspot and need to be amortized by the accelerator to achieve overall speedups. By reducing these data transfer overheads, we are able to improve the overall speedup that can be achieved by the application.

6.8 Chapter Conclusion

In this chapter, we evaluated our RTCS approach from different perspectives. Initially, the heterogeneous node and the different accelerators used in our evaluation platform were introduced. We also introduced the different OpenCL devices and their properties. We took a detailed look at the set of benchmark applications that were extracted from diverse domains. The measuring, data acquisition and the automatic outlier elimination methods were also described in detail. We then looked at the different types of overheads introduced by the RTCS. The overheads introduced by the RTCS before it starts executing the application are known as *application launch overheads* and represent the duration between the time the application is launched (via the RTCS) and the time when the application starts executing on the CPU (also known as latency). On average, we observed that this overhead was around 4 seconds with the JIT compilation phase responsible for most of the overhead. Although this is a one-time overhead, incurred only when the application is launched, it still needs to be considered when executing an application via the RTCS.

We also looked at the *OpenCL code generation and integration overheads* that arose from the OpenCL code generation and integration steps of the RTCS. The compilation of the lambda expression dominates these overheads. Overall, these overheads are quite small (less than 1 second) and do not directly contribute to the application runtime. They can be further reduced by making use of the code caching mechanism of the RTCS Server. The *OpenCL kernel compilation* overhead is introduced by compiling the OpenCL C kernel into device specific binaries. We saw that for hotspots with a large number of kernels, the Nvidia platform was more efficient than the Intel platform. The OpenCL compilation for the PHI was more than 3× slower than the `mCPU` and the `GPGPU`. These overheads however do not directly contribute to the overall execution time of the application and can be reduced by using the code caching mechanism of the RTCS Server. The *application-level OpenCL overheads* are dominated by the overheads introduced by loading the OpenCL kernel binary into the application. For each OpenCL device, these overheads are nearly constant across all benchmark applications. Although these overheads are relatively small, they contribute to the overall execution time of the application and need to be amortized. The overheads introduced by the `mCPU` and `GPGPU` are quite similar, while the `PHI` is more than 4× slower.

Additionally, we looked at the performance improvement of our approach and evaluated the performance at the *kernel-level* as well as at the *hotspot-level* for a representative set of the benchmark applications across different input sizes. We also compared our approach against a pragma-guided OpenACC approach. We saw that the `GPGPU` performed best and was followed by the `PHI` and the `mCPU` in most cases. Comparing our approach against a pragma-guided OpenACC approach we observed that the RTCS kernel-level speedups were comparable to and in most cases better than OpenACC. Looking at all benchmark applications at the hotspot-level, we observed speedups of $9\times$, $25\times$ and $10\times$ for the `mCPU`, `GPGPU` and `PHI` respectively, resulting in a considerable performance improvement over sequential applications.

We evaluated the *tiling performance* and concluded that our tiling approach was most effective for the `GPGPU`, giving us a considerable performance improvement over the generalized approach. We finally looked at the advantages of our *data transfer optimizations* and saw that we were able to save 44% for the `GPGPU` and 23% for the `PHI` in overall data transfer times.

CHAPTER 7

Conclusion and Outlook

In this chapter, we summarize the results of this thesis and discuss about the potential future research opportunities of our work.

7.1 Summary

In this thesis, we introduced and presented our novel automatic and transparent approach known as the Runtime and Just-in-Time Compilation System (RTCS), which is capable of transparently porting sequential programs to different heterogeneous multi-accelerator architectures via OpenCL. We saw how the RTCS is able to analyze loop structures in an application and automatically identify parallelization opportunities, transforming suitable data-parallel loops into independent OpenCL work-items (OpenCL kernels). The RTCS also automatically generates the intricate OpenCL host code required to setup the OpenCL device, create OpenCL buffers, transfer data to the device and launch the OpenCL kernel on the device. We looked at how the RTCS improves the OpenCL kernel performance by applying tiling to improve global memory efficiency. Additionally, the RTCS also applies data transfer optimizations to further improve the application performance.

The RTCS supplements the state-of-the-art SAVE Heterogeneous System Architecture (*saveHSA*), enabling the *saveHSA* to offload computation to different heterogeneous accelerators by employing the RTCS to JIT generate missing accelerator specific implementations. The combination of transparent and flexible code generation support for different target architectures makes the RTCS unique in the domain of parallelization and offloading tools.

We demonstrated that the RTCS can improve the application performance — with no user intervention. To this end, we evaluated the RTCS on a diverse set of benchmark applications from a broad set of domains like scientific computing, security and signal and image processing. We evaluated the RTCS overheads from different perspectives, focusing on *application launch overheads*, *OpenCL code generation and integration overheads*, *OpenCL kernel compilation* and *OpenCL overheads* at the application level. Our evaluation also shows, that the RTCS can achieve comparable performance to handwritten pragma-based OpenACC code, while being fully automated. Looking at the performance of the RTCS across all benchmark applications, we achieve speedups of 9×, 25× and 10× for the `mCPU`, `GPGPU` and `PHI` respectively, resulting in a considerable performance improvement over sequential applications. Additionally, we evaluated the *tiling performance* and saw that our tiling approach was most effective for the `GPGPU` giving us a 9× average performance improvement over the generalized approach. We finally evaluated our *data transfer optimizations* and saw that we were able to save 44% for the `GPGPU` and 23% for the `PHI` in overall data transfer times to and from the accelerator.

7.2 Outlook

In this section, we first take a look at the potential use cases of the RTCS and how the RTCS can be used to automatically target FPGAs. We also look at different promising research directions regarding different OpenCL performance optimizations as well as loop parallelization techniques that can be applied to improve the RTCS.

7.2.1 RTCS Use Cases

In this thesis, we looked at the *saveHSA* and demonstrated how the *saveHSA* benefits from the RTCS's JIT code generation capabilities. We also saw how the structure of a SAVE-Enabled application (SEA) was used to identify hotspots and integrate the generated accelerator implementation at runtime. However, the RTCS has been designed to be modular and is not limited to the *saveHSA* or SEAs. The different RTCS components can be replaced or updated at a later date with state-of-the-art versions. Additionally, by modifying a few components of the RTCS, it can be used in different scenarios. For example, in *HTrOP* [Rie19], we modified a few components of the RTCS to use a polyhedral model (Polly) and target legacy sequential applications instead of SAVE-Enabled applications.

Another scenario where components of the RTCS can be used, is in an automatic legacy-to-OpenCL transformation tool. This tool could be used to transparently convert sequential legacy applications into applications that can execute in parallel on different accelerators that support OpenCL. In this thesis, we present the RTCS as a Just-in-Time compilation approach, however, the RTCS could also be used as a compile-time-only approach. In this approach, the OpenCL transformation tool accepts the legacy application as input and then automatically detects hotspots, parallelizes them, generates OpenCL kernel code and corresponding OpenCL host code as well as integrates it into the legacy application. The output of such an OpenCL transformation tool would be a transformed OpenCL-enabled application binary along with the corresponding pre-compiled OpenCL program.

So far, we have only looked at how applications from the HPC domain benefit from the RTCS. However, the RTCS can also help improve the performance of applications in domains where OpenCL can be used (e.g. embedded systems or mobile devices). Evaluating the impact of the RTCS across these different domains, as well as investigating different domain specific optimizations is an interesting area for future research.

7.2.2 Targeting FPGAs

FPGAs are well suited for HPC workloads, with applications from diverse domains like bioinformatics [64], climate modeling [26], geophysics [58], linear algebra [58] and molecular dynamics [18] performing substantially better on FPGAs as compared to CPUs, and in the case of climate modeling and geophysics, perform substantially better than GPGPUs. However, one major challenge when working with FPGAs is the development process. FPGAs have been mainly programmed by using hardware description languages (HDL) like VHDL and Verilog to model the application on the FPGA. This is a time consuming and tedious process where hardware designers need to have detailed knowledge about the FPGA's low-level building blocks (registers, lookup tables, memory, etc.). Lately, however, this has improved with vendors like Intel and Xilinx releasing tools that allow application developers to program FPGAs using OpenCL [73, 43], making it easier to program FPGAs. This allows software programmers familiar with OpenCL to program FPGAs. Although, this works at a conceptual level, practically, one still needs to have some FPGA knowledge in order to obtain any reasonable performance from the FPGA. Intel and Xilinx are aware of this and have released best practice guides [45] and environment optimization guides [43] to explain the FPGA architecture and provide guidelines that need to be followed to achieve better performance.

We looked at the Intel best practice guide and determined the optimizations and best practices that could prove crucial to the performance of OpenCL kernels on FPGAs. Based on this, we created a prototypical FPGA back-end for the RTCS Server that can target Intel FPGAs. Our FPGA back-end is able to generate single work-item kernels as well as ND-Range kernels. It can automatically maximize FPGA usage by iteratively analyzing the report generated by the `aoc` tool and updating the OpenCL C kernels. Presently, this is an active research area and the FPGA back-end is still under development. However, when combined with an auto-tuning approach, it could prove to be a good framework to investigate how different variants of the same kernel perform on an FPGA.

7.2.3 Optimizing OpenCL Performance

Determining the optimal work-group size: The performance of an OpenCL kernel depends on the size of the work-group. This depends on a number of factors like the OpenCL kernel size as well as the accelerator architecture. As shown by Cummins et al. [20], determining an optimal work-group size can lead to better performance. In our approach, the OpenCL work-group size is not explicitly specified by the RTCS, but is automatically selected by the OpenCL runtime. State-of-the-art methods apply machine learning-based auto-tuning and source code transformation techniques to learn the significant features of the underlying hardware and the kernel implementations. Integrating such methods into the RTCS is an interesting area to further improve the results.

Performance portability: OpenCL is functionally portable across different accelerator architectures. It achieves this by abstracting away the different accelerators as *OpenCL devices*, allowing the developer to target different accelerators with the same OpenCL code. However, as seen in our evaluation, this functional portability does not guarantee performance portability across different accelerator architectures. Accelerator specific optimizations are required to improve the OpenCL performance for different devices. Different code transformation as well as auto-tuning techniques presented in related work [81, 35, 76, 75] have shown to be promising in generating highly optimized architecture specific code. This is an interesting research direction to help improve the performance of the RTCS across different accelerators.

7.2.4 Automatic Loop Parallelization

Auto parallelization of sequential applications is a challenging task. In our work, we demonstrate that our parallelization approach is able to achieve comparable performance to handwritten pragma-based OpenACC code. In order to guarantee the correctness of the parallelization, our approach only considers natural loops without any cross-iteration data dependencies. However, Bondhugula et al. [15] demonstrate that in many cases, loops with data dependencies can be transformed to make them parallel. Integrating such loop transformation techniques into the RTCS would allow the RTCS to target a more diverse range of applications.

7.3 Chapter Conclusion

In this chapter, we looked at the different and interesting future research directions that can be taken to build upon and improve upon the RTCS. We also saw that the RTCS can not only be used in model based approaches like the *saveHSA*, but can also be adapted to accelerate general applications (demonstrated by *HTrOP*). Users are able to automatically accelerate legacy applications from diverse domains and offload computation to different accelerators without any additional effort or interaction from the application developer. Not only does this result in improved application performance, but it also reduces porting effort required by the application developer. This in turn translates into savings in development time as well as costs, making the RTCS highly relevant in the domain of parallelization and offloading tools.

Acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree
C++ AMP	C++ Accelerated Massive Parallelism
CPU	Central Processing Unit
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
GPGPU	General-Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HDS	Hotspot Data Structure
HPC	High Performance Computing
HSA	Heterogeneous System Architecture
IDE	Integrated Development Environment
IQR	Interquartile Range
LLVM IR	LLVM Intermediate Representation
ispc	Intel SPMD Program Compiler
JIT	Just-in-Time
MC	Machine Code
MCJIT	Machine Code Just-In-Time
mCPU	Multi-core CPU

MIC Many Integrated Cores

NVPTX Nvidia Parallel Thread Execution

ODA Observe-Decide-Act

OpenCL Open Computing Language

ORC MCJIT On-Request Compilation Machine Code Just-In-Time

QoS Quality of Service

RPC Remote Procedure Call

RTCS Runtime and Just-in-Time Compilation System

SAVE SAVE Self-Adaptive Virtualization-aware high-performance/low-Energy heterogeneous system architecture

saveHSA SAVE Heterogeneous System Architecture

SCoPs Static Control Parts

SEA SAVE-Enabled application

SLP Superword-Level Parallelism

SoC Systems-on-Chip

SPMD Single Program Multiple Data

SSA Static Single Assignment

SUIF Stanford University Intermediate Format

TBB Threading Building Blocks

VaR Value at Risk

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

VM Virtual Machine

Author's Publications

[Rie19] Heinrich Riebler, Gavin Vaz, Tobias Kenter, and Christian Plessl. Transparent acceleration for heterogeneous platforms with compilation to opencl. *ACM Trans. Archit. Code Optim.*, 16(2):14:1–14:26, April 2019. ISSN 1544-3566. doi: 10.1145/3319423.

[Rie18] Heinrich Riebler, Gavin Vaz, Tobias Kenter, and Christian Plessl. Automated code acceleration targeting heterogeneous opencl devices. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 417–418. ACM, New York, NY, USA, 2018. ISBN 978-1-4503-4982-6. doi:10.1145/3178487.3178534.

[Vaz16] Gavin Vaz, Heinrich Riebler, Tobias Kenter, and Christian Plessl. Potential and methods for embedding dynamic offloading decisions into application code. *Comput. Electr. Eng.*, 55(C):91–111, October 2016. ISSN 0045-7906. doi: 10.1016/j.compeleceng.2016.04.021.

[Rie16] Heinrich Riebler, Gavin Vaz, Christian Plessl, Ettore M. G. Trainiti, Gianluca C. Durelli, Emanuele Del Sozzo, Marco D. Santambrogio, and Cristiana Bolchini. Using just-in-time code generation for transparent resource management in heterogeneous systems. In *2016 IEEE 2nd International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI)*, pages 1–5. Sep. 2016. doi:10.1109/RTSI.2016.7740545.

[Dam15] Marvin Damschen, Heinrich Riebler, Gavin Vaz, and Christian Plessl. Transparent offloading of computational hotspots from binary code to Xeon Phi. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pages 1078–1083. EDA Consortium, March 2015.

[Vaz14] Gavin Vaz, Heinrich Riebler, Tobias Kenter, and Christian Plessl. Deferring accelerator offloading decisions to application runtime. In *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE Computer Society, December 2014. Received best paper award.

[Dur14] Gianluca Durelli, Marcello Pogliani, Antonio Miele, Christian Plessl, Heinrich Riebler, Gavin Vaz, Marco D. Santambrogio, and Cristiana Bolchini. Runtime resource management in heterogeneous system architectures: The save approach. In *2014 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 142–149. Aug 2014. ISSN 2158-9178. doi:10.1109/ISPA.2014.27.

[Ken14] Tobias Kenter, Gavin Vaz, and Christian Plessl. Partitioning and vectorizing binary applications for a reconfigurable vector computer. In *Reconfigurable Computing: Architectures, Tools, and Applications*, pages 144–155. Springer International Publishing, Cham, 2014. ISBN 978-3-319-05960-0.

[Ram13] Franz Rammig, Katharina Stahl, and Gavin Vaz. A framework for enhancing dependability in self-x systems by artificial immune systems. In *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*, pages 1–10. June 2013. ISSN 1555-0885. doi:10.1109/ISORC.2013.6913240.

Bibliography

[1] Saman P. Amarasinghe, Jennifer M. Anderson, Monica S. Lam, and Chau wen Tseng. An Overview of the SUIF Compiler for Scalable Parallel Machines. In *In Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 662–667. 1993.

[2] AMD APP SDK: OpenCL User Guide. *URL: http://developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_User_Guide2.pdf.* [Online; accessed 26-Febuary-2019].

[3] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A Compiler-level Intermediate Representation Based Binary Analysis and Rewriting System. In *Proc. ACM European Conference on Computer Systems (EuroSys)*, EuroSys '13, pages 295–308. ACM, 2013. ISBN 978-1-4503-1994-2. doi:10.1145/2465351.2465380.

[4] Aditi Athavale, Priti Ranadive, M. N. Babu, Prasad Pawar, Sudhakar Sah, Vinay Vaidya, and Chaitanya Rajguru. Automatic Sequential to Parallel Code Conversion. *GSTF Journal on Computing (JoC)*, 1(4), 2018. ISSN 2010-2283.

[5] Auto-Vectorization in LLVM. *URL: https://llvm.org/docs/Vectorizers.html.* [Online; accessed 04-July-2019].

[6] Utpal Banerjee. An introduction to a formal theory of dependence analysis. *The Journal of Supercomputing*, 2(2):133–149, Oct 1988. ISSN 1573-0484. doi:10.1007/BF00128174.

[7] Cédric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16. Juan-les-Pins, France, September 2004.

[8] B. Betkaoui, D. B. Thomas, and W. Luk. Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing. In *Int. Conf. on Field-Programmable Technology (ICFPT)*, pages 94–101. Dec 2010. doi:10.1109/FPT.2010.5681761.

[9] Aart Bik, Milind Girkar, Paul Grey, and X. Tian. Efficient Exploitation of Parallelism on Pentium III and Pentium 4 Processor-Based Systems, 2001.

[10] Alecio Pedro Delazari Binotto, Dionisio Doering, Thorsten Stetzelberger, Patrick McVittie, Sergio Zimmermann, and Carlos Eduardo Pereira. A CPU, GPU, FPGA system for X-ray image processing using high-speed scientific cameras. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2013 25th International Symposium on*, pages 113–119. IEEE, 2013.

[11] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *The journal of political economy*, pages 637–654, 1973.

[12] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, and T. Lawrence. Parallel programming with Polaris. *Computer*, 29(12):78–82, Dec 1996. ISSN 0018-9162. doi:10.1109/2.546612.

[13] C. Bolchini, G. C. Durelli, A. Miele, G. Pallotta, and M. D. Santambrogio. An orchestrated approach to efficiently manage resources in heterogeneous system architectures. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 200–207. Oct 2015. doi:10.1109/ICCD.2015.7357104.

[14] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *Int. Conf. on Compiler Construction (ETAPS CC)*. April 2008.

[15] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *SIGPLAN Not.*, 43(6):101–113, June 2008. ISSN 0362-1340. doi:10.1145/1379022.1375595.

[16] H. Bryhni, E. Klovning, and O. Kure. A comparison of load balancing techniques for scalable Web servers. *IEEE Network*, 14(4):58–64, July 2000. ISSN 0890-8044. doi:10.1109/65.855480.

[17] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009. ISSN 0167-739X. doi:http://dx.doi.org/10.1016/j.future.2008.12.001.

[18] Matt Chiu and Martin C. Herbordt. Molecular Dynamics Simulations on High-Performance Reconfigurable Computing Systems. *ACM Trans. Reconfigurable Technol. Syst.*, 3(4):23:1–23:37, November 2010. ISSN 1936-7406. doi:10.1145/1862648.1862653.

[19] C++ AMP : Language and Programming Model. [Online; accessed 02-May-2019].

[20] Chris Cummins, Pavlos Petoumenos, Michel Steuwer, and Hugh Leather. Autotuning OpenCL workgroup size for stencil patterns. *arXiv preprint arXiv:1511.02490*, 2015.

[21] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[22] C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *Computer*, 42(12):36–42, Dec 2009. ISSN 0018-9162. doi:10.1109/MC.2009.385.

[23] G. Durelli, M. Coppola, K. Djafarian, G. Kornaros, A. Miele, M. Paolino, Oliver Pell, Christian Plessl, M. D. Santambrogio, and C. Bolchini. SAVE: Towards Efficient Resource Management in Heterogeneous System Architectures. In Diana Goehringer, Marco Domenico Santambrogio, João M. P. Cardoso, and Koen Bertels, editors, *Reconfigurable Computing: Architectures, Tools, and Applications*, pages 337–344. Springer International Publishing, Cham, 2014. ISBN 978-3-319-05960-0.

[24] Gianluca C. Durelli and Marco D. Santambrogio. Autonomic Thread Scaling Library for QoS Management. *SIGBED Rev.*, 13(1):41–47, March 2016. ISSN 1551-3688. doi:10.1145/2907972.2907978.

[25] Fast ISPC Texture Compressor. *URL: `https://software.intel.com/en-us/articles/fast-ispc-texture-compressor`*. [Online; accessed 02-April-2019].

[26] Lin Gan, Haohuan Fu, Wayne Luk, Chao Yang, Wei Xue, Xiaomeng Huang, Youhui Zhang, and Guangwen Yang. Solving the Global Atmospheric Equations Through Heterogeneous Reconfigurable Platforms. *ACM Trans. Reconfigurable Technol. Syst.*, 8(2):11:1–11:16, March 2015. ISSN 1936-7406. doi:10.1145/2629581.

[27] Oliver Gay. C++ SHA256 Function. *URL: `http://www.zedwood.com/article/cpp-sha256-function`*. [Online; accessed 04-July-2019].

[28] Google Test. *URL: `https://github.com/google/googletest`*. [Online; accessed 04-July-2019].

[29] Ivan Grasso, Petar Radojkovic, Nikola Rajovic, Isaac Gelado, and Alex Ramirez. Energy Efficient HPC on Embedded SoCs: Optimization Techniques for Mali GPU. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 123–132. IEEE, 2014.

[30] Johan Gronqvist and Anton Lokhmotov. Optimising OpenCL kernels for the ARM Mali-T600 GPUs. *GPU Pro 5: Advanced Rendering Techniques*, page 327, 2014.

[31] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel computing*, 22(6):789–828, 1996.

[32] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly-Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[33] Tobias Grosser and Torsten Hoefler. Polly-ACC Transparent Compilation to Heterogeneous Hardware. In *Proc. Int. Conf. on Supercomputing*, page 1. ACM, 2016.

[34] OpenACC Working Group et al. The OpenACC Application Programming Interface. *URL: `https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf`*, 2011. [Online; accessed 04-July-2019].

[35] Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, and Bixia Zheng. Twin Peaks: A Software Platform for Heterogeneous Computing on General-purpose and Graphics Processors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 205–216. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0178-7. doi: 10.1145/1854273.1854302.

[36] Markus Happe, Friedhelm Meyer auf der Heide, Peter Kling, Marco Platzner, and Christian Plessl. On-The-Fly Computing: A Novel Paradigm for Individualized IT Services. In *Proc. Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*. IEEE Computer Society, June 2013.

[37] Mark Harris. Mini-Nbody: A Simple N-body Code. *URL: `https://github.com/harrism/mini-nbody`*, 2014. [Online; accessed 04-July-2019].

[38] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 79–88. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0074-2. doi: 10.1145/1809049.1809065.

[39] HSA Foundation. *URL: `http://www.hsafoundation.com/`*. [Online; accessed 24-May-2019].

[40] Heterogeneous Systems Architecture Foundation Launches HSA 1.1 Specification with Multi-Vendor Architecture Support. *URL: `http://www.hsafoundation.com/heterogeneous-systems-architecture-foundation-launches-hsa-1-1-specification-multi-vendor-architecture-support/`*. [Online; accessed 24-May-2019].

[41] HTrOP Repository. *URL: `https://pc2.uni-paderborn.de/research/publications/open-source-projects/automated-code-acceleration-with-compilation-to-opencl/`*. [Online; accessed 25-May-2019].

[42] Xilinx Inc. SDAccel Environment Optimization Guide. *URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug1207-sdaccel-optimization-guide.pdf.* [Online; accessed 21-June-2019].

[43] Xilinx Inc. SDAccel Programmers Guide. *URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1277-sdaccel-programmers-guide.pdf*, 2019. [Online; accessed 04-July-2019].

[44] Intel C++ Compiler 19.0 Developer Guide and Reference. *URL: https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-automatic-parallelization.* [Online; accessed 22-May-2019].

[45] Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide. *URL: https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807516407.html.* [Online; accessed 21-June-2019].

[46] Intel Xeon Phi Coprocessor. *URL: https://software.intel.com/en-us/articles/opencl-design-and-programming-guide-for-the-intel-xeon-phi-coprocessor.* [Online; accessed 04-July-2019].

[47] Stephen Johnson, Emyr Evans, Haoqiang Jin, and Constantinos Ierotheou. The ParaWise Expert Assistant – Widening Accessibility to Efficient and Scalable Tool Generated OpenMP Code. In Barbara M. Chapman, editor, *Shared Memory Parallel Programming with Open MP*, pages 67–82. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-31832-3.

[48] Lester Kalms, Tim Hebbeler, and Diana Göhringer. Automatic OpenCL Code Generation from LLVM-IR using Polyhedral Optimization. In *Proc. Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, pages 45–50. ACM, 2018.

[49] Khronos OpenCL Registry. *URL: https://www.khronos.org/registry/OpenCL.* [Online; accessed 11-March-2019].

[50] W. Kim and M. Voss. Multicore Desktop Programming with Intel Threading Building Blocks. *IEEE Software*, 28(1):23–31, Jan 2011. ISSN 0740-7459. doi: 10.1109/MS.2011.12.

[51] Stephen Kokoska and Daniel Zwillinger. *CRC Standard Probability and Statistics Tables and Formulae.* CRC Press, 1999.

[52] Lambda expressions. *URL: https://en.cppreference.com/w/cpp/language/lambda.* [Online; accessed 15-January-2019].

[53] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. Int. Symp. Code Generation and Optimization*, pages 75–86. 2004.

[54] D. Lenoski, J. Laudon, K. Gharachorloo, W. . Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–79, March 1992. ISSN 0018-9162. doi:10.1109/2.121510.

[55] Jyrki Leskela, Jarmo Nikula, and Mika Salmela. OpenCL embedded profile prototype in mobile device. In *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, pages 279–284. IEEE, 2009.

[56] Chunhua Liao, Daniel J. Quinlan, Jeremiah J. Willcock, and Thomas Panas. Extending Automatic Parallelization to Optimize High-Level Abstractions for Multicore. In Matthias S. Müller, Bronis R. de Supinski, and Barbara M. Chapman, editors, *Evolving OpenMP in an Age of Extreme Parallelism*, pages 28–41. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-02303-3.

[57] Shih-Wei Liao, Amer Diwan, Robert P. Bosch, Jr., Anwar Ghuloum, and Monica S. Lam. SUIF Explorer: An Interactive and Interprocedural Parallelizer. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, pages 37–48. ACM, New York, NY, USA, 1999. ISBN 1-58113-100-3. doi:10.1145/301104.301108.

[58] O. Lindtjorn, R. Clapp, O. Pell, H. Fu, M. Flynn, and O. Mencer. Beyond Traditional Microprocessors for Geoscience High-Performance Computing Applications. *IEEE Micro*, 31(2):41–49, March 2011. ISSN 0272-1732. doi:10.1109/MM.2011.17.

[59] LLVM Analysis and Transform Passes. *URL: https://llvm.org/docs/Passes.html*. [Online; accessed 04-July-2019].

[60] The LLVM Compiler Infrastructure Project. *URL: https://llvm.org/*. [Online; accessed 04-July-2019].

[61] LLVM Execution Engine. *URL: http://llvm.org/doxygen/classllvm_1_1ExecutionEngine.html*. [Online; accessed 11-March-2019].

[62] LLVM's Analysis and Transform Passes. *URL: https://llvm.org/docs/Passes.html*. [Online; accessed 18-October-2018].

[63] Alberto Magni, Christophe Dubach, and Michael O'Boyle. Automatic Optimization of Thread-coarsening for Graphics Processors. In *Proc. Int. Conf. on Parallel Architectures and Compilation (PACT)*, PACT '14, pages 455–466. ACM, New York, NY, USA, 2014. ISBN 978-1-4503-2809-8. doi:10.1145/2628071.2628087.

[64] Atabak Mahram and Martin C. Herbordt. NCBI BLASTP on High-Performance Reconfigurable Computing Systems. *ACM Trans. Reconfigurable Technol. Syst.*, 7(4):33:1–33:20, January 2015. ISSN 1936-7406. doi:10.1145/2629691.

[65] Simon Moll. *Decompilation of LLVM IR.* Master's thesis, Saarland University, 2011.

[66] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, 2008.

[67] Sebastian Nilsson. FIR Filter Arduino Library. *URL:* `https://github.com/sebnil/FIR-filter-Arduino-Library`, 2017. [Online; accessed 04-July-2019].

[68] Par4All. *URL:* `http://par4all.github.io/`. [Online; accessed 23-May-2019].

[69] PGI Compilers & Tools. *URL:* `https://www.pgroup.com/index.htm`. [Online; accessed 22-May-2019].

[70] Louis-Noël Pouchet. Polybench: The Polyhedral Benchmark Suite. *URL:* `http://web.cs.ucla.edu/~pouchet/software/polybench/`, 2018. [Online; accessed 04-July-2019].

[71] Gabriel Rivera and Chau-Wen Tseng. A Comparison of Compiler Tiling Algorithms. In Stefan Jähnichen, editor, *Compiler Construction*, pages 168–182. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-49051-7.

[72] Save's Goals. *URL:* `http://www.fp7-save.eu/project.htm`. [Online; accessed 02-April-2019].

[73] Deshanand Singh. Implementing FPGA Design with the OpenCL Standard. *Altera whitepaper*, 2011.

[74] 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, 2010.

[75] John A. Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen-mei W. Hwu. Efficient Compilation of Fine-grained SPMD-threaded Programs for Multicore CPUs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 111–119. ACM, New York, NY, USA, 2010. ISBN 978-1-60558-635-9. doi:10.1145/1772954.1772971.

[76] John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In José Nelson Amaral, editor, *Languages and Compilers for Parallel Computing*, pages 16–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-89740-8.

[77] The open standard for parallel programming of heterogeneous systems. *URL:* `https://www.khronos.org/opencl/`. [Online; accessed 28-May-2019].

[78] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a Holistic Approach to Auto-parallelization: Integrating Profile-driven Parallelism Detection and Machine-learning Based Mapping. *SIGPLAN Not.*, 44(6):177–187, June 2009. ISSN 0362-1340. doi:10.1145/1543135.1542496.

[79] User Guide for NVPTX Back-end. *URL:* `https://llvm.org/docs/NVPTXUsage.html`. [Online; accessed 04-July-2019].

[80] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.

[81] Mei Wen, Da-fei Huang, Chang-qing Xun, and Dong Chen. Improving performance portability for GPU-specific OpenCL kernels on multi-core/many-core CPUs by analysis-based transformations. *Frontiers of Information Technology & Electronic Engineering*, 16(11):899–916, Nov 2015. ISSN 2095-9230. doi:10.1631/FITEE.1500032.

[82] Trent Willis. Advanced Ray-Tracer. *URL:* `https://github.com/trentmwillis/ray-tracer`, 2014. [Online; accessed 04-July-2019].

[83] Writing an LLVM Pass. *URL:* `http://llvm.org/docs/WritingAnLLVMPass.html`. [Online; accessed 18-October-2018].

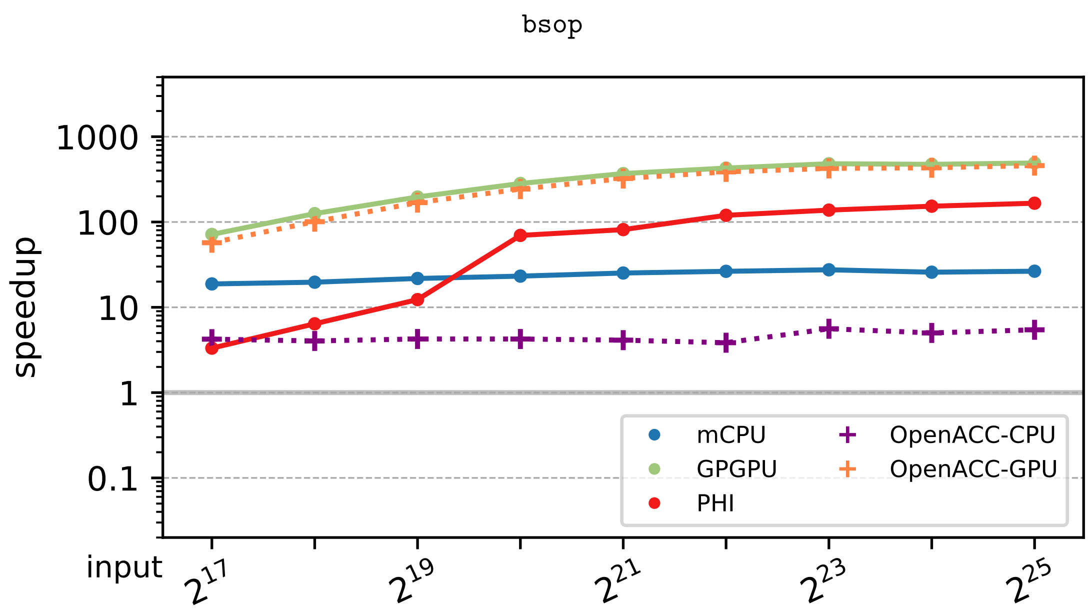
[84] Erik Wynters. Fast and Easy Parallel Processing on GPUs Using C++ AMP. *J. Comput. Sci. Coll.*, 31(6):27–33, June 2016. ISSN 1937-4771.

[85] OpenCL NVIDIA Developer Zone. NVIDIA OpenCL SDK Code Samples. *URL:* `https://developer.nvidia.com/opencl`, 2018. [Online; accessed 04-July-2019].

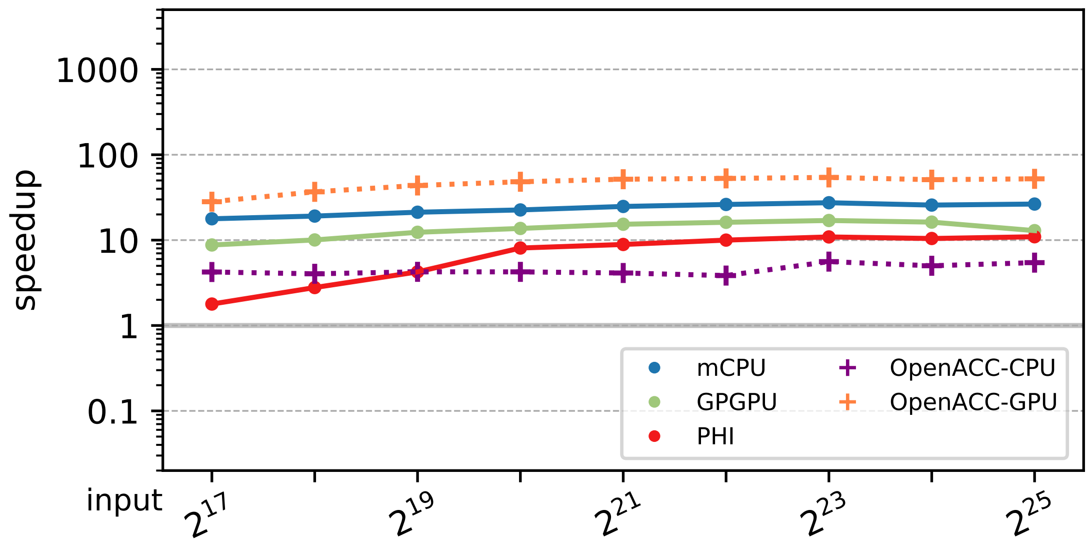
APPENDIX A

Kernel and Hotspot-level speedups

This appendix includes all kernel-level and hotspot-level speedups that have not been presented in the performance evaluation section (Section 6.5) of the thesis.

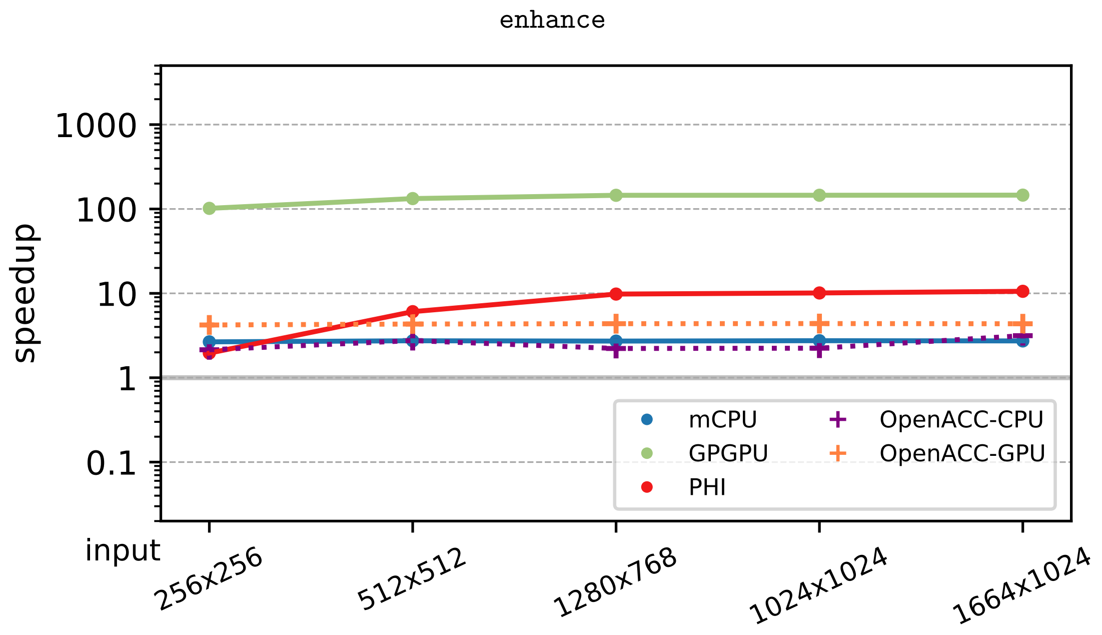


(a) Kernel.

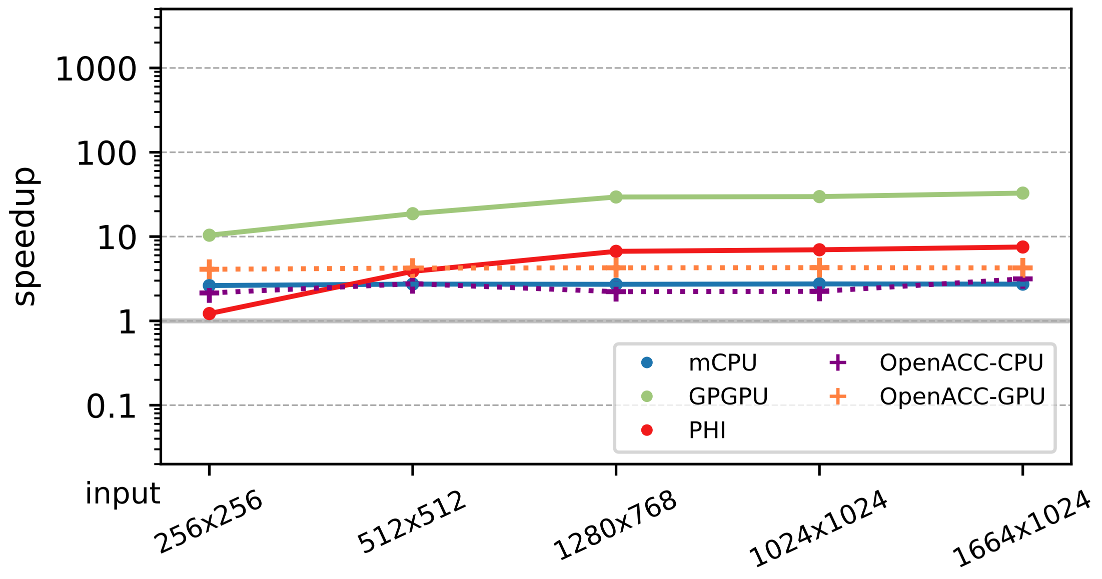


(b) Hotspot including data transfer.

Figure A.1: `bsop`: Kernel and hotspot-level speedup for different input sizes. Plot (a) shows the kernel-level speedup, while plot (b) shows the hotspot-level speedup including data transfer overheads.

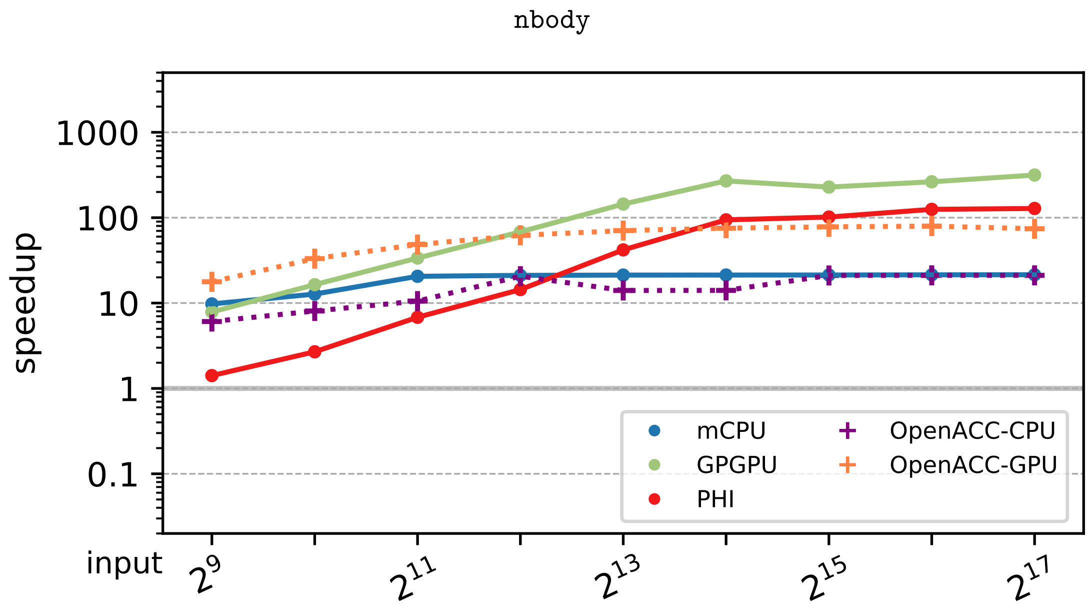


(a) Kernel.

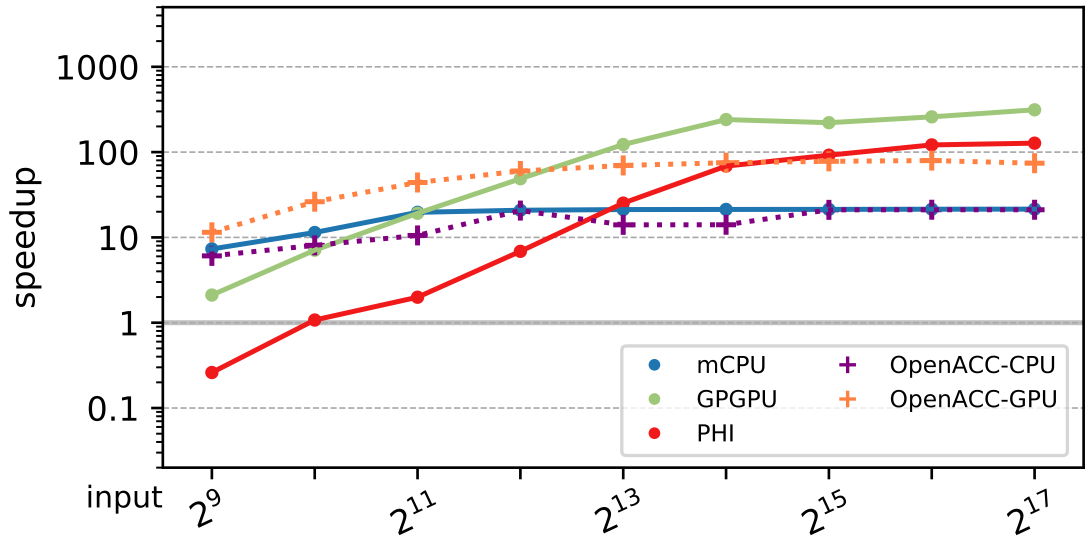


(b) Hotspot including data transfer.

Figure A.2: `enhance`: Kernel and hotspot-level speedup for different input sizes. Plot (a) shows the kernel-level speedup, while plot (b) shows the hotspot-level speedup including data transfer overheads.

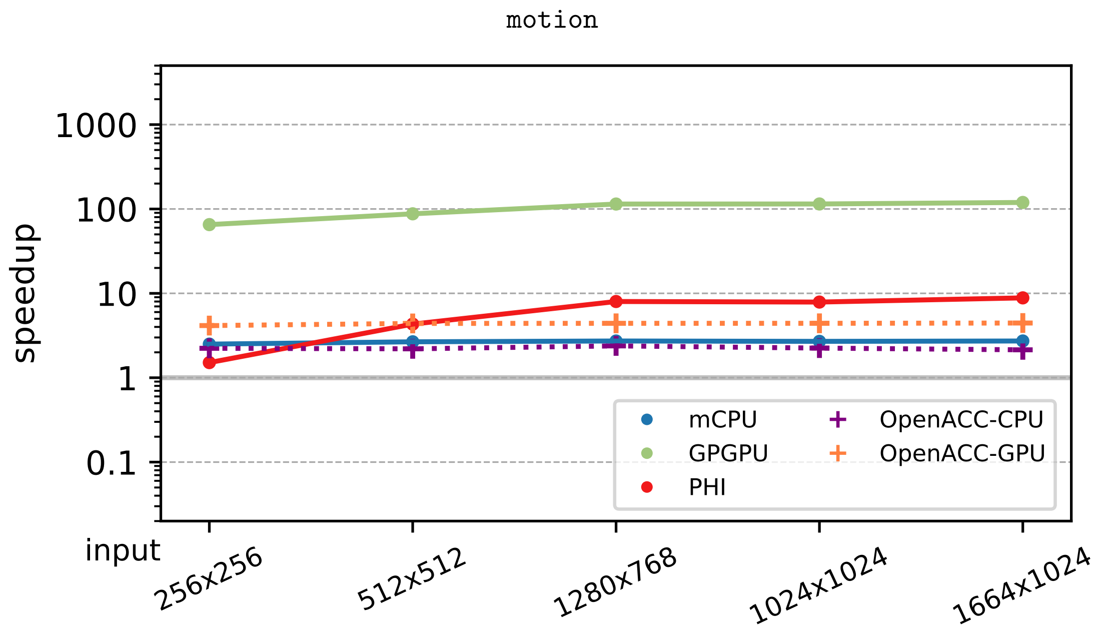


(a) Kernel.

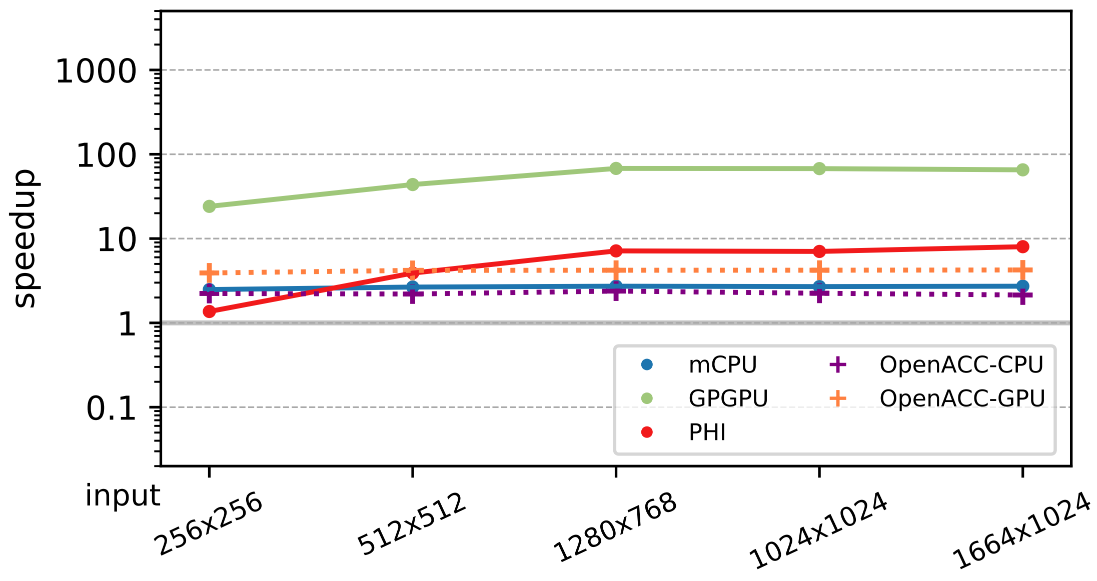


(b) Hotspot including data transfer.

Figure A.3: `nbody`: Kernel and hotspot-level speedup for different input sizes. Plot (a) shows the kernel-level speedup, while plot (b) shows the hotspot-level speedup including data transfer overheads.

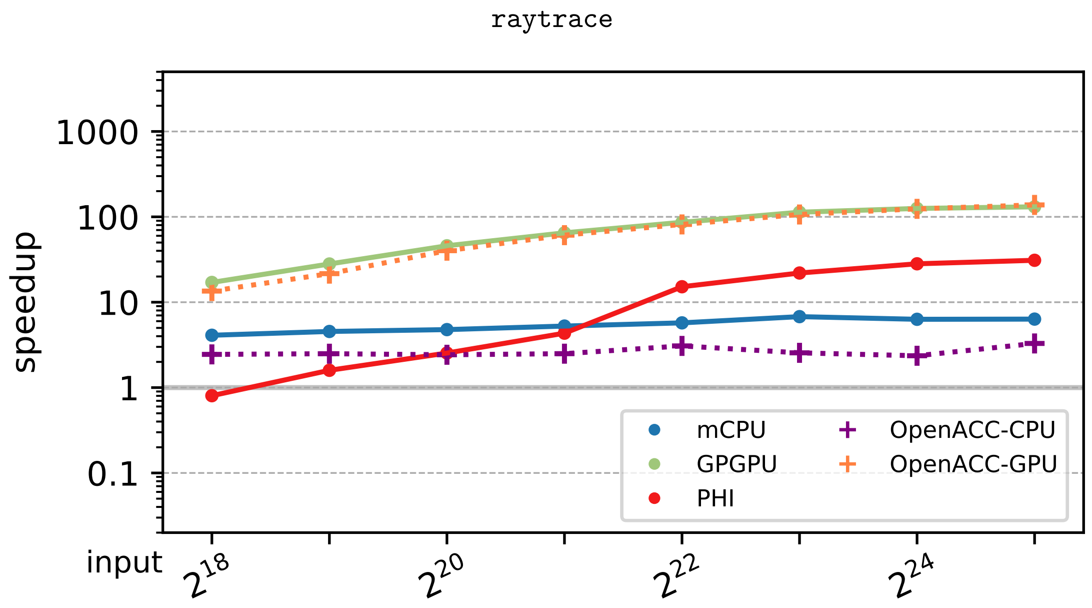


(a) Kernel.

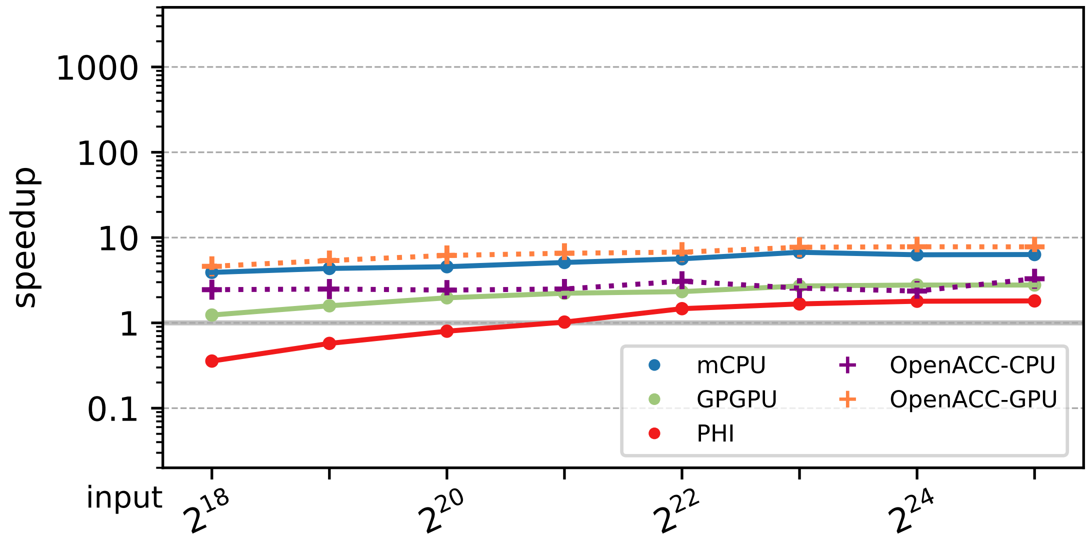


(b) Hotspot including data transfer.

Figure A.4: `motion`: Kernel and hotspot-level speedup for different input sizes. Plot (a) shows the kernel-level speedup, while plot (b) shows the hotspot-level speedup including data transfer overheads.

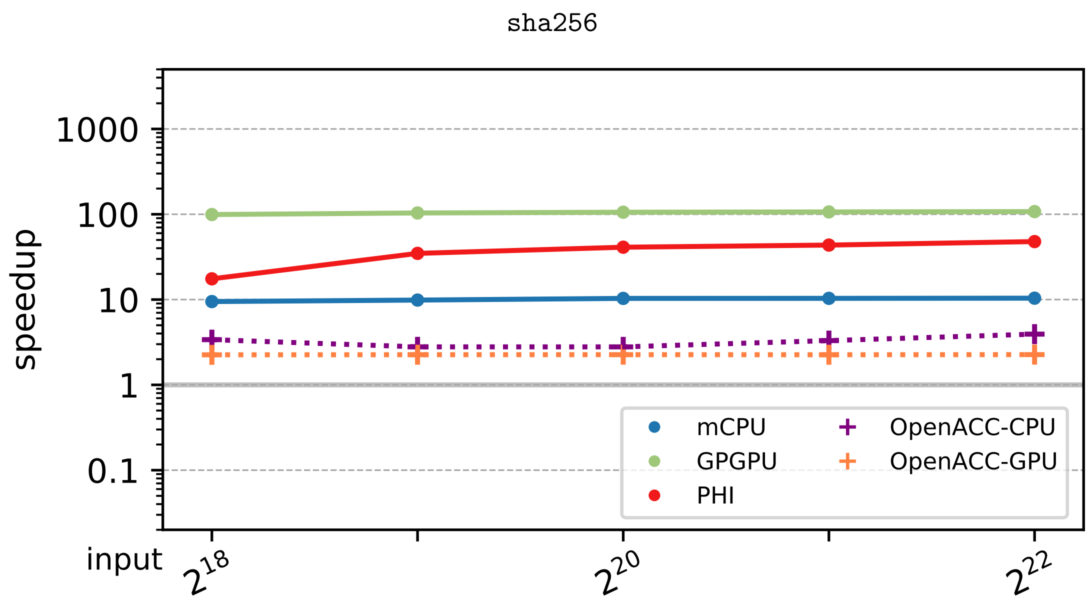


(a) Kernel.

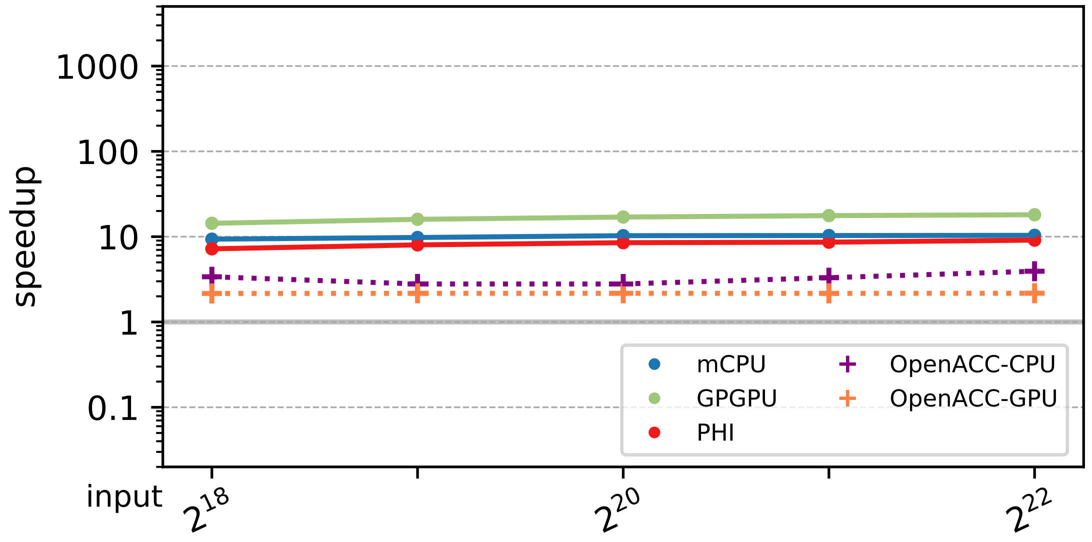


(b) Hotspot including data transfer.

Figure A.5: `raytrace`: Kernel and hotspot-level speedup for different input sizes. Plot (a) shows the kernel-level speedup, while plot (b) shows the hotspot-level speedup including data transfer overheads.

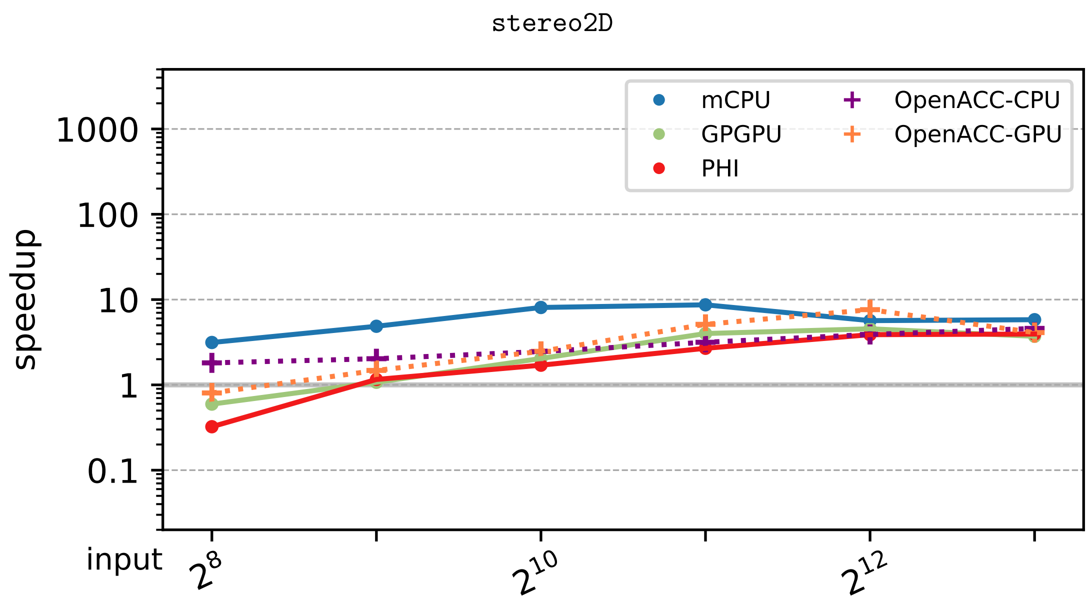


(a) Kernel.

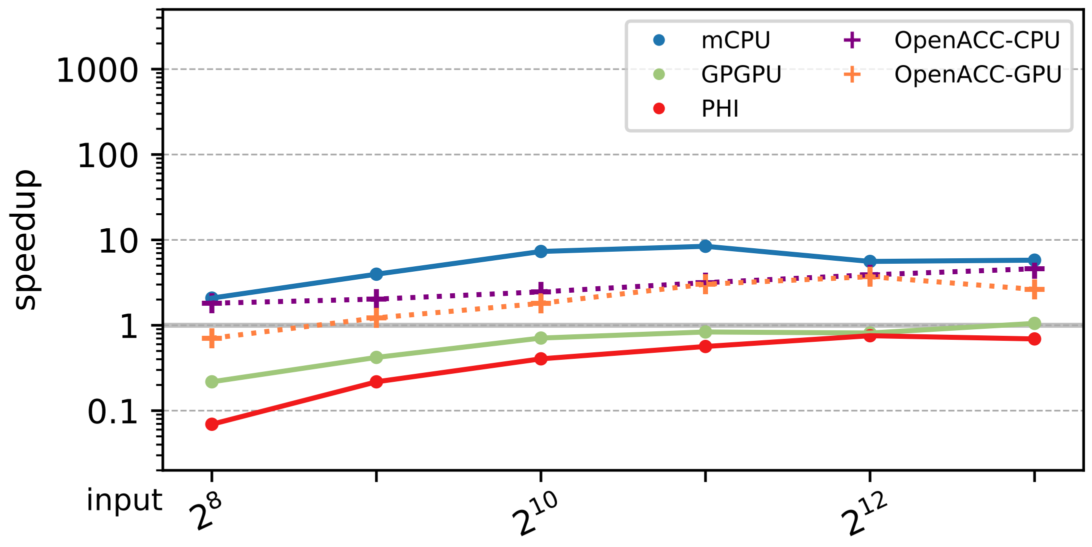


(b) Hotspot including data transfer.

Figure A.6: `sha256`: Kernel and hotspot-level speedup for different input sizes. Plot (a) shows the kernel-level speedup, while plot (b) shows the hotspot-level speedup including data transfer overheads.



(a) Kernel.



(b) Hotspot including data transfer.

Figure A.7: `stereo2D`: Kernel and hotspot-level speedup for different input sizes. Plot (a) shows the kernel-level speedup, while plot (b) shows the hotspot-level speedup including data transfer overheads.