Efficient High Performance Computing on Heterogeneous Platforms

Jie Shen

Efficient High Performance Computing on Heterogeneous Platforms

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op
dinsdag 24 november 2015 om 12:30 uur

door

Jie SHEN

Bachelor of Engineering,
National University of Defense Technology, China

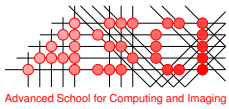
geboren te Changsha, China

Dit proefschrift is goedgekeurd door de:

Promotor: Prof.dr.ir. H.J. Sips
Copromotor: Dr.ir. A.L. Varbanescu

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof.dr.ir. H.J. Sips	Technische Universiteit Delft, promotor
Dr.ir. A.L. Varbanescu	Technische Universiteit Delft & Universiteit van Amsterdam, copromotor
Onafhankelijke leden:	
Prof.dr. M. Huisman	Universiteit Twente
Prof.dr. A. Lastovetsky	University College Dublin
Prof.dr. K.L.M. Bertels	Technische Universiteit Delft
Prof.dr.ir. H.X. Lin	Technische Universiteit Delft & Universiteit Leiden
Overig lid:	
Dr. X. Martorell	Universitat Politècnica de Catalunya



This work was carried out in the ASCI graduate school. ASCI dissertation series number 341.

This work was supported by the China Scholarship Council (CSC).

Part of this work has been done in collaboration with the Barcelona Supercomputing Center, and supported by a HiPEAC collaboration grant.

Published and distributed by: Jie Shen
E-mail: jieshen.hetcomp@gmail.com
ISBN: 978-94-6203-954-4

Cover design: the cover, designed by Jie Shen and produced by Yating Liao and Jiankun Tang, uses the “bulb and brain” concept to represent the smart ideas behind efficient heterogeneous computing with the CPU and ACCelerators. Some of the cover images are from http://www.58pic.com/.
Printed in the Netherlands by: Wöhrmann Print Service.

Throughout the ages all those who have been highly successful in great ventures and in the pursuit of learning must of necessity have [successively] experienced three kinds of state (ching-chieh).

Last night the west wind shrivelled the green-clad trees,
Alone I climb the high tower
To gaze my fill along the road to the horizon.

expresses the first state.

My clothes grow daily more loose, yet care I not.
For you am I thus wasting away in sorrow and pain.

expresses the second state.

I sought her in the crowd a hundred, a thousand times.
Suddenly with a turn of the head [I saw her],
That one there where the lamplight was fading.

expresses the third state.

—Wang Kuo-Wei, *Jen-Chien Tz'u-Hua*
(translated by Adele Austin Rickett)

Acknowledgements

Four years ago, I flew around 9,000 kilometres to the Netherlands—for the first time, I left China and home, and started the adventure of pursuing a PhD. Now, after I have completed this thesis and look back, I find that I am so lucky to have met many people, who gave me a hand to hold, an ear to listen, and a heart to understand. I would like to express my gratitude to all of you. Without you, I could not have been where I am today.

My deepest thanks go to my supervisors, Dr. Ana Lucia Varbanescu and Prof. Henk Sips. Ana, thank you for your inspirational guidance, consistent support, and warm encouragement. You have been motivating me to become an independent researcher with enthusiasm and confidence, and you are a good role model to me. I value the free, friendly and also vigorous research environment you have built for me (and your other students) that stimulates innovative ideas and productive outcomes. I appreciate your patience and dedication in reviewing my articles without neglecting even a single "the", in meeting me at TU Delft every week although you have moved to UvA two years ago, and in replying to my emails no matter how early in the morning, late at night, or in the weekends. I am happy to be your PhD student (to be more specific, the first girl PhD student). Working with you has been a pleasant and invaluable experience to me.

Henk, thank you for offering me a PhD position in the PDS group. Without your approval and support, I could not have been able to pursue my PhD abroad and I could not have come here to have these wonderful four years. I appreciate your visionary guidance throughout my PhD research and your full support (with a gentle smile all the time on your face) that makes me progress smoothly here. I enjoy our casual conversations on widely different subjects that broaden my horizons, and I have been always impressed by your great insight and sense of humor. I hope your plan to have an academic visit to China will come true sometime in the near future.

I would like to thank Prof. Yutong Lu, my bachelor and master supervisor in China, for bringing me to the field of parallel and distributed computing, and for encouraging me to pursue a PhD. Your encouragement and words have magic power that makes me believe in myself whenever I am in an impasse. Prof. Peng Zou, thank you for your guidance and support during my years in NUDT. You are a kind and inspiring supervisor, who have shown me how to be energetic both in and outside academia.

I would like to thank Dr. Michael Arntzen, my co-author of one important publication in my PhD research. Michael, thank you for bringing me an interesting and challenging project that has helped me find a niche in the field of heterogeneous computing and that has lead to the first version of the Glinda framework. I enjoyed this excellent collaboration and exchanging physics and computer science knowledge with you.

Dr. Xavier Martorell, thank you for hosting me for a 3-month visit to the Barcelona Supercomputing Center. Our work on different workload partitioning strategies has helped me understand much more about task- and data-level parallelism on heterogeneous platforms. I appreciate our collaboration, and I hope to extend it in the future.

Dr. Alexandru Iosup, thank you for helping me with my scientific writing and presentation, for teaching me the important components that must be in a good research paper, and for including me in your research group at the end of my PhD.

I would like to thank my committee members for the assessment of my draft thesis and for the valuable comments to help me improve. I would like to thank Prof. Lina Sarro for being my mentor in the Graduate School. I also want to thank China Scholarship Council for my PhD scholarships.

Many thanks to my colleagues and friends in the PDS group.

Jianbin, thank you for involving me in your work when I started my PhD, which helped me to a good start. I also have enjoyed our discussions ranging from parallelism and performance to ideals and life.

Bogdan, thank you for being a good friend and officemate. You, Yong and I have started PhD studies at roughly the same time, and we have shared many ups and downs in these four years. And thank you and Elena for the new year gift and greetings in Chinese.

Alexander, thank you for your valuable insights on OpenCL programming and behavior (especially zero copy on CPUs) when I worked on my first paper.

Adele, thank you for sharing useful tips on everyday life in the Netherlands and for inviting me to the new year dinner every year at your home with a beautiful view of fireworks afterwards. I wish you a happy life, full of joy, with your baby boy.

Siqi, I admire your critical mind and outgoing character, and I know that behind it there is a very warm heart, always ready to help.

Otto, thank you for your help with all Dutch related matters. You always give me a quick and easy solution. Thank you and Ernst for translating the summary of this thesis into Dutch. You, too, I wish a happy life with your new family.

Alexey, it was a happy tour with you, Dick, and Vincent in CCGrid'15 and China Folk Culture Villages in Shenzhen. I wish you a great time during your PhD.

Wing, Lipu, Yunhua, Kefeng, Siqi, and Yong, thank you for building and maintaining a small China town in your (previous) office, where we have a lot of fun talks and "serious" debates.

Elric, thank you for helping me to prepare my visit to Barcelona and for introducing

me to the interesting events there. Ana (U. Valladolid), Jorge, Moisés, thank you for visiting our group and sharing your research interests with me.

Thank you, Lucia, Dimitra, Boxun, Nitin, Mihai, Niels, Rahim, Riccardo, Tamás, Boudewijn, Arno, Alex (small) for monthly dinner, for birthday party, for badminton, and for our lunch chats. Without you, I could not have so many wonderful memories in the PDS group.

Paulo, Munire, and Stephen, thank you for the excellent technical support that allows me to focus on my work. Ilse, Rina, Shemara, Franca, and Monique, thank you for taking care of the administrative issues. It is nice knowing you all.

To my friends in the Netherlands, I owe you a big thank you.

Xiaoqin, thank you for sharing your life, work, and memories of Changsha with me, and for the yummy food and beautiful knits you made. I am amazed by your nimble fingers and thoughtfulness, and I have learned a lot from you.

Shuai, we have been "neighbors" since 2007, from neighboring dormitory at NUDT to living in the same street in Delft. We have many common topics and ideas. Thank you for being by my side and cheering me up all the time.

Minxin, I am happy that I have got a friend like you, generous, reliable, and brave. I am sure that your future business and life will have lots of success and joy.

Thank you, Hao, Wei, Yao, Yongchang, Lixia, Jinglang, Tiantian, Xinchao, Shanshan, Yanqing, and other NUDTers in the Netherlands (sorry, I cannot write down all your names) for brightening up my spare time.

I am also grateful to my friends in China. Yating, thank you for being my friend since elementary school, and thank you and your boyfriend for transforming my fuzzy idea into an actual thesis cover. Thank you, Benlan, Yuan, Hu, Ling, and Huajian, for your constant support and encouragement, and for your hospitality during my holiday in China. And thank Hu for taking care of my administrative issues in NUDT. Dan, Haidong, Xinye, Chengye, Hao, thank you for our random talks over WeChat and Skype which are always full of laughter. I wish you all the best.

My special thanks to Yong, thank you for sharing happiness and success, and for overcoming difficulties and challenges with me. I appreciate your company more than you will ever know.

Finally, I want to express my sincerest gratitude and love to my parents. Thank you for bringing me to the world, for raising me up, for teaching me, and for giving me all your love. You are the best parents in the world. This thesis is dedicated to you.

Jie
Delft, October 2015

Contents

Chapter 1

Introduction

In the scientific world, the demand for higher performance never stops. Various applications from different fields rely on high performance computing (HPC) to accomplish the computation goal, to speedup the research development cycle, and to expand the research boundaries. Traditional scientific applications, like weather forecasting, computational fluid dynamics, and physics simulation, require their jobs to be finished as soon as possible. Emerging applications in fields like material science, bioinformatics, and medical science rely exclusively on HPC solutions. With such solutions, the studies in these fields can be realized, and eventually bring benefit to our daily life. To push the boundaries of HPC, parallel and distributed computing have often been employed on a high-end, "luxury" solution. However, as the hardware technology advances, parallel computing becomes ubiquitous, and HPC machines are becoming collections of many-core nodes, each of which has its own heterogeneous structure.

In this context, we define heterogeneous platforms as mixes of different types of processors in a compute node or a chip package. By contrast, homogeneous platforms only have one type of processors. By integrating processors with distinct hardware capabilities, heterogeneous platforms have the potential to improve application performance [45, 87]. Over the last decade, the development of multi-core CPUs and many-core accelerators, such as GPUs (Graphical Processing Units) [89] and Intel MICs (Many Integrated Cores) [57], has promoted the progression and prevalence of heterogeneous platforms. Combining the CPU and the accelerator at the node (e.g., CPU+GPU platforms) or chip level (e.g., Intel Sandy Bridge and its successors [23, 136], AMD APUs—Accelerated Processing Units [15]) becomes an attractive and worthwhile option in computer systems ranging from supercomputers to mobile devices [132, 135]. Looking ahead, heterogeneous platforms will keep gaining popularity, so their efficiency and usability must be improved. Combining platform heterogeneity and application diversity, we obtain a large search space, and therefore achieving high performance on such platforms is an inherently complex problem. In this thesis, we aim to develop systematic methods for a wide

spectrum of parallel applications to efficiently utilize heterogeneous platforms.

Heterogeneous platforms have manifested their advantages over homogeneous platforms in several important aspects. One major advantage is performance. By making good use of different types of processors, heterogeneous platforms allow parallel applications to achieve greater speedup than homogeneous platforms [1, 88]. For instance, assuming we have a platform with one GPU and one CPU, the GPU with many "small and simple" cores can process application workloads with massive parallelism and regular behavior, while the CPU with a few "large and complex" cores can process sequential workloads and workloads with less parallelism and more irregularity. Another advantage is power efficiency. Whereas the CPUs are facing the power wall, accelerators such as GPUs and FPGAs (Field Programmable Gate Arrays) are designed to offer great power efficiency [19, 49]. In addition, the hardware mix also allows for more flexible application execution. An application can be accelerated on either one of the processors, or all processors based on certain performance and/or power criteria [43, 48]. In this thesis, we focus on the performance advantage with the aim to achieve the best performance on heterogeneous platforms.

Processor disparity poses significant challenges to utilizing heterogeneous platforms. First, multiple programming models have been designed to exploit different types of processors, but which programming model(s) are efficient to program heterogeneous platforms remains an open question. Second, as the processors have distinct hardware characteristics, how to partition the application workload and assign each partition to the most suitable processor is not trivial. In addition to the hardware challenges, the application challenges make this problem even more complex. We must be aware that processors can perform widely different based on the application to execute. Even the same application can have very different performance with different datasets. This diversity of applications and datasets must be taken into account when partitioning the application workload. In this thesis, we follow an application-centric approach to study high performance heterogeneous computing. Specifically, we focus on data parallel applications as they are mainstream applications that demand high performance. We select a unified programming model for heterogeneous platforms, and improve its usability to fully exploit the underlying processors. We design, develop, test, and tune workload partitioning methods that allow us to harvest the performance of heterogeneous platforms. In the end, we propose a framework that enables a large variety of data parallel applications to achieve the best performance on heterogeneous platforms.

1.1 Data Parallel Applications

Data parallel applications have massive data parallelism. Previously mentioned applications in various scientific fields are all data parallel applications which can be accelerated

on multi-core and many-core machines for achieving high performance. The data independence is the key feature of data parallel applications. In the parallelization dimension (see Figure 1.1), the data points of the parallelization space are independent of each other (e.g., pixels in a medical image, options in a pricing model), and therefore all the data points can be computed in parallel without interdependencies.

To understand the workload of data parallel applications, we consider a data parallel application in three dimensions illustrated in Figure 1.1. The first dimension is the parallelization dimension, i.e., the data point dimension. The second dimension is the workload dimension, where data points can have balanced (uniform) workload, or imbalanced (varied) workload. Whether the workload is balanced or imbalanced is determined by the application algorithm and/or dataset. The third dimension is the kernel dimension. A kernel is a section of code that has data parallelism. An application may have one or multiple kernels connected in a certain execution flow, so we take this dimension into account for applications with more complex kernel structures.

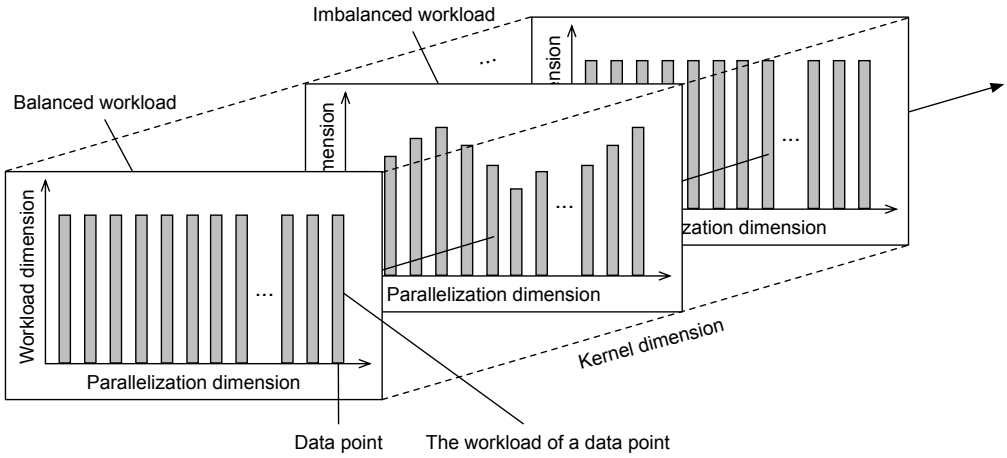


Figure 1.1: A data parallel application in three dimensions: the parallelization dimension, the workload dimension, and the kernel dimension.

In this thesis, we gradually increase the number of dimensions we focus on: we start from the parallelization dimension to understand how to program different processors on heterogeneous platforms; next, we include both parallelization and workload dimensions to explore efficient workload partitioning over heterogeneous platforms; finally, we consider all three dimensions together to further improve the applicability of our framework.

1.2 Parallel Programming Models

A parallel programming model is an abstraction of the underlying hardware that allows an application to express parallelism. It forms a bridge between the application parallelism

and the hardware architecture [7]. Multiple programming models have been proposed to exploit multi-core and many-core processors. According to the level of portability, we classify them into two categories: dedicated programming models and unified programming models.

Dedicated programming models are proposed only for a specific type of processors. For example, OpenMP [94] is proposed to target multi-core CPUs. It allows programmers to parallelize the sequential C, C++, and Fortran code by adding compiler directives. The annotated code can be executed on shared memory multi-core machines. CUDA [90] is also a dedicated programming model which is specifically designed for Nvidia GPUs. CUDA provides a low-level exploitation of Nvidia hardware architecture, so the CUDA code cannot be ported to GPUs from other vendors (e.g., AMD GPUs) or other types of processors (e.g., multi-core CPUs and Intel MICs).

Unified programming models are proposed for cross-platform portability. Programming models belonging to this category can be used for different types of processors with little porting effort. OpenCL [64] is the first open standard programming model designed to take full advantage of heterogeneous platforms. Code written in OpenCL are portable across a diverse mix of multi-core CPUs, Nvidia and AMD GPUs, Intel MICs, ARM based SoCs, FPGAs, and DSPs. OpenMP 4.0 [95], OpenACC [93], and OmpSs [11] extend the original concept of OpenMP by supporting new compiler directives for accelerator programming. In these models, programmers specify loops and regions of host code to be offloaded to an accelerator processor. These models provide high-level programming, as they hide hardware architecture details from the programmers. Compared to them, OpenCL provides the programmers with low-level control of the underlying hardware.

In this thesis, we choose a unified programming model to exploit heterogeneous platforms in a unified way.

1.3 Heterogeneous Platforms

In this section, we introduce the hardware model of heterogeneous platforms, the platform composition, and the platforms that we empirically used to build our work on.

1.3.1 Host-Accelerator Hardware Model

Heterogeneous platforms are organized by the host-accelerator model. Figure 1.2 shows the overview of the hardware model. In this model, one or multiple accelerators (regarded as devices) are connected to the host, and the host manages the accelerators to perform the application workload in a collaborative way. On heterogeneous platforms, the (multi-core) CPU acts as the host, while the accelerators can be different types, such as multi-core CPUs, GPUs, Intel MICs, and FPGAs.

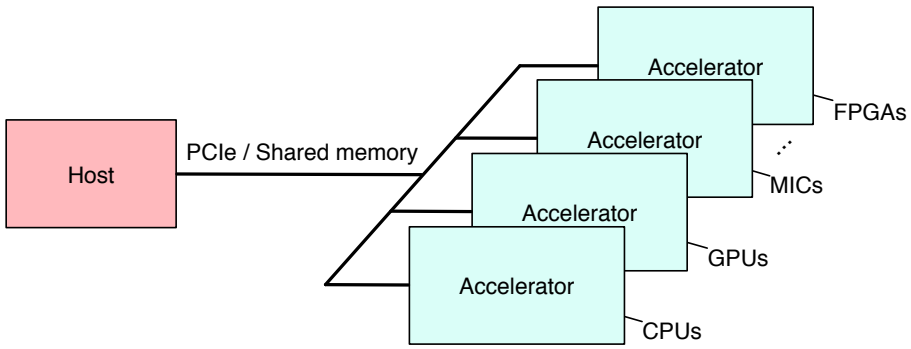


Figure 1.2: The host-accelerator hardware model.

The host performs the sequential part of the application, and offloads the parallel workload to the accelerators. To enable the offloading, the host must make the data that needs to be computed accessible to the accelerators before the computation starts, and must gather the data back after the computation finishes. This host-accelerator data communication can be achieved with two kinds of physical connections, which are PCIe (PCI Express) bus and shared memory. In the case of the PCIe connection, the host and the accelerators have separate memories. Data needs to be explicitly transferred between the memories, and the data transfer speed is determined by the PCIe bandwidth. Heterogeneous platforms with discrete accelerators, such as discrete GPUs and Intel MICs, are using the PCIe connection model. In the case of the shared memory connection, the host and the accelerators are integrated in the same chip and share the same memory. Because of the integrated architecture, the host does not need to perform a real data transfer, and the data coherence is maintained by the hardware. Typical examples of the shared memory connection can be found in Intel Sandy Bridge and its successors, AMD APUs, and ARM based mobile SoCs (System on a Chip).

1.3.2 Multi-cores and Many-cores

The most widely used accelerators on heterogeneous platforms are multi-core and many-core processors. The two classes of processors have become the main trends in microprocessor development over the last ten years [66]. Mainly due to the power consumption problem, traditional single-core processors cannot provide the applications with continuous performance improvement by increasing the clock rate. Instead, the microprocessor vendors switched the strategy by adding more cores on a single chip [126].

Multi-core processors have multiple cores (usually 2–32 cores), and offer multiple layers of parallelism [31, 74]. Besides instruction level parallelism, the multiple cores together with SMT (Simultaneously Multi-Threading) technology offer thread level parallelism as well. For example, Intel's Hyper-Threading [69] allows each core to implement

two hardware threads simultaneously, so a quad-core CPU can operate eight hardware threads. In addition, the use of vector units and a vector instruction set enable data level parallelism, known as SIMD (Single Instruction Multiple Data) parallelism. On multi-core processors, the memory system is organized into a hierarchy by adding multiple levels of on-chip caches between the registers and the off-chip memory. This design reduces the memory access latency for many applications whose performance is bounded by memory accesses. Typical multi-core processors are general-purpose multi-core CPUs. Two or more multi-core CPUs can be plugged into CPU sockets to enhance the CPU computing power, resulting in a NUMA (Non-Uniform Memory Access) architecture [82].

Many-core processors integrate many more cores (usually hundreds or thousands of cores) compared to multi-core processors, but each core is smaller and simpler [22,57,77]. This hardware design allows many-core processors to utilize SIMT (Single Instruction Multiple Threads) parallelism and to offer high computation throughput. In other words, although the time to process one instruction is longer than multi-core processors, the overall cost can be amortized by housing a large number of threads. Such high degree of multi-threading effectively hides memory access latency and makes many-core processors suitable for massively parallel applications. There are also multiple levels of caches (usually two levels instead of three levels on multi-core CPUs) and programmable scratch-pad memories that applications can use to maximize performance. Typical many-core processors are GPUs and Intel MICs. GPUs were originally used as pure graphics accelerators, and are now extensively used for general-purpose GPU processing (known as GPGPU). Intel MIC (brand name, Xeon Phi) was launched into the market in late 2012, and is growing in popularity in data parallel applications.

1.3.3 CPU+GPU Heterogeneous Platforms

Among various forms of heterogeneous platforms, we empirically build our work on the CPU+GPU heterogeneous platforms. Our choice is determined based on the following observations.

On one hand, GPUs cannot work in a standalone mode. The host, which is a multi-core CPU[1], is needed to manage the GPU execution and the host-accelerator communication. The multi-core CPU is free for use when the GPU executes the application workload. As the CPU has multiple layers of parallelism, it can be used as an accelerator as well. That is to say, CPU+GPU platforms have native heterogeneity to exploit. On the other hand, GPUs have been widely used in general-purpose computing. How to implement and optimize parallel applications on GPUs has been extensively studied [16, 58, 70, 106]. As a result, there is a large collection of GPU code available. If the GPU code can be ported to CPUs and the ported code can perform well on CPUs, we have a large collection of

[1] In this thesis, we regard multi-socket CPUs as an entire CPU, i.e., one accelerator.

code for heterogeneous computing almost for free. Therefore, it is worthwhile to start from CPU+GPU heterogeneous platforms, and to understand how we can efficiently utilize both processors to achieve the best application performance. As we develop generic and systematic methods, our methods can also be applied to other types of heterogeneous platforms that match the host-accelerator hardware model.

1.4 Problem Statement

Heterogeneous platforms are ubiquitous, and their efficient use becomes a key factor in the performance behavior of many data parallel applications, especially as we move towards larger scale datasets. This efficient use depends significantly on the target applications and available hardware resources. In this thesis, we investigate systematic, application-centric methods to enable efficient high performance computing on heterogeneous platforms. Our research is driven by the following questions.

RQ1: How to efficiently use OpenCL programming model on heterogeneous platforms? OpenCL is a unified programming model: the same code can be executed correctly on different types of processors. OpenCL support is currently available for CPUs (Intel, AMD, and ARM), GPUs (Nvidia, AMD, and ARM), APUs (AMD), MICs (Intel), and FPGAs (Altera and Xilinx). This cross-platform code portability makes OpenCL an interesting option for heterogeneous computing, and it is the main reason for which we choose to use OpenCL for programming the applications presented in this thesis. However, we must also investigate how efficient OpenCL is, in terms of performance, for heterogeneous platforms, because significant performance losses due to portability are not acceptable in our performance-driven research.

RQ2: What is a good solution to accelerate imbalanced applications? An important class of applications that require acceleration are massively parallel imbalanced applications. These applications can be found in domains of scientific simulation, numerical methods, and graph processing [129], where relatively few data points in the parallelization space require more computation than other data points, resulting in an imbalanced workload. The imbalanced workload can severely diminish the hardware utilization of a homogeneous platform. Intuitively, it seems beneficial to accelerate imbalanced applications on heterogeneous platforms, where a smart workload partitioning can match the heterogeneity of the platform with the imbalance of the workload. Therefore, it is essential to determine how to maximize performance by partitioning the workload over the heterogeneous platform.

RQ3: How to optimize the workload partitioning process? Achieving the best performance on heterogeneous platforms is only possible when the workload is partitioned to best utilize all the heterogeneous components. Obtaining such an optimal workload partitioning is not trivial: the characteristics of the application workload, the capabilities

of the hardware processors, and the data transfer between the host and the processors must be taken into account. An auto-tuning based method ensures partitioning optimality, but usually requires multiple rounds until reaching the best partitioning. Therefore, we investigate the feasibility of a model-based method for optimal workload partitioning.

RQ4: How to generalize workload partitioning to balanced applications? Compared to imbalanced applications, balanced applications have regular, uniform workloads. It seems reasonable to accelerate balanced applications on homogeneous many-core platforms. However, due to the accelerator's memory capacity, the application's parallelism degree, and the data transfer overhead, homogeneous many-core platforms such as GPUs may not always be a wise option. Therefore, it is necessary to investigate how to accelerate balanced applications on heterogeneous platforms by generalizing workload partitioning. To this end, we need a comprehensive study to understand balanced applications' performance behavior, and a systematic approach to determine the best hardware configuration on the platform and the optimal workload partitioning if necessary.

RQ5: How to maximize the performance and applicability for workload partitioning? So far, we have considered data parallel applications to only be characterized by their workload balance or imbalance. Now we consider applications that have one or multiple kernels executed in a certain execution flow. Both static and dynamic workload partitioning strategies exist on heterogeneous platforms, but their applicability and performance differ significantly depending on the application to execute. Therefore, it is beneficial to design a unified workload partitioning method that satisfies both requirements—the performance and the applicability—in one go. A promising design should have a right classification of applications, a set of partitioning strategies, and a matchmaking policy that matches applications and partitioning strategies for efficient execution on heterogeneous platforms.

1.5 Contributions and Thesis Outline

This thesis is divided into 7 chapters. Figure 1.3 shows the structure of the thesis. The contributions of the thesis are as follows:

In Chapter 2, **OpenCL as A Programming Model for Heterogeneous Platforms**, we demonstrate the efficiency of OpenCL as a programming model for heterogeneous platforms. We study how OpenCL can be efficiently used on heterogeneous platforms by focusing on its performance portability. Originating from the GPGPU world, OpenCL has been widely used and proved to behave well on GPUs [29], but its performance on CPUs remains unsolved. In this context, we study the factors that impact OpenCL performance when porting the code from GPUs to CPUs. Specifically, we use the performance of regular OpenMP code as a reference, and gradually tune the OpenCL code to match it. By quantifying the performance impact of each tuning step, we isolate those significant is-

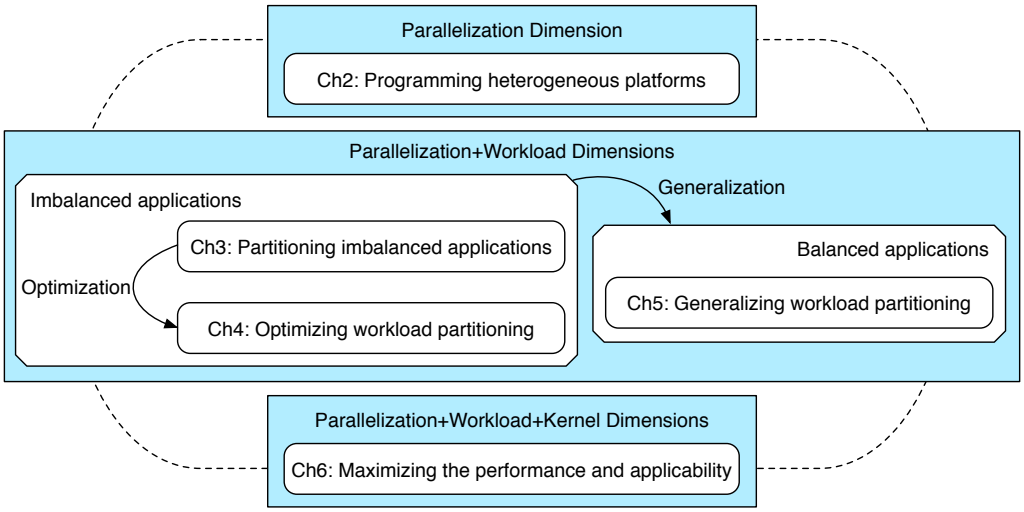


Figure 1.3: Structure of the thesis.

sues. Based on the identified performance impact factors, we propose systematic methods to use OpenCL on CPUs and to transform OpenCL code between CPU-friendly and GPU-friendly forms. Our study improves OpenCL's efficiency for programming heterogeneous platforms. This chapter is largely based on our paper [111]:

- **Jie Shen**, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. An application-centric evaluation of OpenCL on multi-core CPUs. *Parallel Computing*, 39(12):834–850, 2013.

In Chapter 3, **Accelerating Imbalanced Applications on Heterogeneous Platforms: A Workload Partitioning Framework**, we propose Glinda, a workload partitioning framework for accelerating imbalanced applications. In this framework, we design and develop mechanisms to detect the workload characteristics, to partition the workload to fit the usage patterns of the processors, and to obtain the optimal workload partitioning that maximizes the performance gain. We use an auto-tuning method to determine the optimal workload partitioning: the workload is decomposed into CPU and GPU tasks, which are tuned (1) in granularity, to find the optimal data parallelism in OpenCL, and (2) in size, to achieve a perfect execution overlap. We empirically demonstrate the effectiveness of our Glinda framework. This chapter is largely based on our paper [118]:

- **Jie Shen**, Ana Lucia Varbanescu, Henk Sips, Michael Arntzen, and Dick G. Simons. Glinda: A Framework for Accelerating Imbalanced Applications on Heterogeneous Platforms. In *Proceedings of the ACM International Conference on Computing Frontiers (CF'13)*, 2013.

In Chapter 4, **Optimizing Workload Partitioning: A Prediction Method**, we propose a model-based partitioning method that optimizes the partitioning process for imbalanced applications. Specifically, we develop a prediction method that replaces the auto-tuning method with a quick and correct prediction of the workload partitioning. The prediction method is built based on modeling the execution of the partitioned workload on heterogeneous platforms. Given a fitting criteria, we build a partitioning model that represents the optimal workload partitioning. On the application side, we use a workload model to quantify the application workload, its workload characteristics, and the data transfer between the host and the accelerators. On the platform side, we estimate the accelerators' hardware capabilities by using a low-cost profiling. Combining these quantities into the partitioning model, we solve the optimal partitioning. We show the accuracy, effectiveness, and adaptiveness of our prediction method with both synthetic benchmarks and real-life applications. This chapter is largely based on our paper [120]:

- **Jie Shen**, Ana Lucia Varbanescu, Peng Zou, Yutong Lu, and Henk Sips. Improving Performance by Matching Imbalanced Workloads with Heterogeneous Platforms. In *Proceedings of the 28th International Conference on Supercomputing (ICS'14)*, 2014.

In Chapter 5, **Generalizing Workload Partitioning: A Systematic Approach**, we demonstrate a generalized model-based method for workload partitioning that improves the performance of both balanced and imbalanced applications. We study the performance behavior of balanced applications, understanding the importance of using the right hardware configuration on the heterogeneous platform. By extending the prediction method developed for imbalanced applications, we propose a systematic approach to determine (1) the best hardware configuration and, when needed, (2) the optimal workload partitioning for accelerating balanced applications. We further generalize our approach to applications with different datasets and platforms with multiple accelerators. This chapter is largely based on our papers [117, 119]:

- **Jie Shen**, Ana Lucia Varbanescu, and Henk Sips. Look before You Leap: Using the Right Hardware Resources to Accelerate Applications. In *Proceedings of the 16th IEEE International Conference on High Performance Computing and Communications (HPCC'14)*, 2014.

- **Jie Shen**, Ana Lucia Varbanescu, Yutong Lu, Peng Zou, and Henk Sips. Efficient Deployment of Data Parallel Applications on Heterogeneous Platforms. *Under review*.

In Chapter 6, **Maximizing the Performance and Applicability for Workload Partitioning**, we propose a unified workload partitioning method that covers single- and multi-kernel data parallel applications. Specifically, to make workload partitioning feasible for

applications with multiple kernels and more complex kernel structures, our novel method is designed to combine the best features of static and dynamic partitioning. We define an application classification based on the analysis of the application kernel structure. We propose five different partitioning strategies and define their performance ranking for each application class. We further design an application analyzer that matches the best partitioning strategy to a given application by determining its class and selecting the best ranked strategy for that class. Combining both static and dynamic partitioning, our method enables a large variety of data parallel applications to be executed efficiently on heterogeneous platforms. This chapter is largely based on our paper [116]:

- **Jie Shen**, Ana Lucia Varbanescu, Xavier Martorell, and Henk Sips. Matchmaking Applications and Partitioning Strategies for Efficient Execution on Heterogeneous Platforms. In *Proceedings of the 44th International Conference on Parallel Processing (ICPP'15)*, 2015.

In Chapter 7, **Conclusions and Future Work**, we present a summary of our findings and, more importantly, we discuss the architecture of a framework that could alleviate the practical limitations of the approaches presented in this thesis, allowing a unified, user-transparent workload partitioning for *any* data parallel application.

Chapter 2

OpenCL as A Programming Model for Heterogeneous Platforms

In this chapter, we evaluate the OpenCL programming model and its suitability for heterogeneous computing. Because of its many similarities with the CUDA programming model, which is designed for NVIDIA GPUs, OpenCL has been mainly used for GPU computing and proved to be able to perform as well as CUDA [29]. This chapter focuses on the potential that OpenCL has to exploit the performance of multi-core CPUs. Specifically, we identify the key performance impact factors for using OpenCL on CPUs and propose a systematic method to adapt OpenCL code to CPUs.

As multi-core CPU programming and GPU programming keep gaining popularity for parallel computing, an open standard programming model, OpenCL (Open Computing Language) [63], has been designed to exploit different types of hardware platforms and facilitate heterogeneous computing. While OpenCL has been studied and proved popular for GPUs, its cross-platform portability makes it an interesting option for programming multi-core CPUs as well.

OpenCL proposes to tackle many-core diversity in a unified way: a common hardware model. The user programs on the OpenCL "virtual" platform, and the resulting source code is portable on any platform that supports OpenCL. Currently, most hardware vendors (e.g., AMD, Apple, ARM, IBM, Intel, NVIDIA) have developed drivers, runtimes, and compilers to support OpenCL on their processors. OpenCL provides the user with a low-level, fine-grained control of the application parallelism. The user is able to control each data point of the parallelization space in the code, and decompose the whole parallelization space to data point groups in a flexible way.

On the other hand, OpenMP (Open Multi-Processing) [10], specifically designed for shared memory parallel machines, remains of interest for multi-core CPUs, because it is very ease to use. In OpenMP, the programmers enable parallel execution by annotating

sequential C, C++, or Fortran code with compiler directives. Sequential algorithms are parallelized incrementally without major restructuring. An OpenMP program operates in the fork-join model. The parallelism granularity in OpenMP can be controlled manually by adjusting the number of OpenMP threads in combination with a scheduling approach, such as `static` or `dynamic`.

Since OpenMP and OpenCL are different in their parallelism approach, and given that OpenMP is *the* programming model for shared memory parallel machines for high-performance computing, we investigate if OpenCL can achieve reasonable performance on CPUs using the OpenMP performance as a reference. Our initial tests comparing Rodinia benchmarks [17] implemented in OpenMP and OpenCL have shown that the performance difference between the two programming models can be very diverse [113]. This chapter focuses on understanding the causes that lead to this divergent performance behavior from the perspective of OpenCL. Our goal is to find the key performance impact factors, if any, that make OpenCL perform better or worse than OpenMP.

This study is of interest to the OpenCL community. First, because GPU programming in OpenCL and CUDA (Compute Unified Device Architecture) [90] continues to gain popularity in parallel computing, OpenCL performance tuning on GPUs [29] and code translation between the two programming models [26] have been widely studied. As a result, there is a large amount of OpenCL GPU code available [17, 24, 124]. As OpenCL has the advantage of code portability, the available code can be directly executed on CPUs, preserving functional correctness and gaining performance through parallelization.

Second, further evaluating the performance of the OpenCL code on CPUs benefits heterogeneous platforms integrating CPUs and GPUs. If the OpenCL code can be tuned to be CPU-friendly and meet the performance of a regular CPU parallel programming model (e.g., OpenMP), the extra effort for developing and maintaining independent CPU and GPU solutions of the same application might be spared. Both CPUs and GPUs can share the same parallelism approach. The workload partitioning [67, 118] and/or scheduling [8, 35] can be managed in a more flexible and systematic way.

Third, OpenCL gives the users more control to tune the application parallelism through its hardware abstraction. In contrast, OpenMP is a high-level parallelism approach. We find that this low-level control in OpenCL can be an advantage when parallelizing applications on CPUs.

Moreover, we notice that all major CPU vendors have announced that they support the OpenCL specification [64] and released their OpenCL SDKs. They are committed to developing their OpenCL implementations to expose the potential of their CPUs. Thus, industry's continuous support and investment will further promote the use of OpenCL on CPUs.

The three interesting points mentioned above are built on the premise that OpenCL can get reasonable performance on multi-core CPUs. Therefore, analyzing and locating

the performance impact factors for OpenCL applications running on CPUs is of critical importance. In our work, we use the performance of *regular parallel OpenMP code* (i.e., not aggressively optimized) as a reference. We gradually tune the OpenCL code, evaluating the performance impact of each tuning step. In this way, we are able to isolate and quantify those significant issues that should not be neglected when looking for well performing OpenCL code on CPUs. We note that similar approaches are also adopted in [26, 29, 68]. While these studies evaluated the performance portability of OpenCL on various GPUs, our work focuses on the CPU side, evaluating the performance impact factors in OpenCL for CPUs and the performance portability between GPUs and CPUs. Moreover, the novelty of our work is not the approach itself, but the results (the understanding of the OpenCL performance on multi-core CPUs) that we obtained.

We evaluate five applications from the Rodinia benchmark suite [17]: K-means, Pathfinder, HotSpot, CFD, and BFS. For all five applications, the original OpenCL code cannot compete with the performance achieved by the regular OpenMP code in most cases (the details on the OpenMP implementations and their performance are discussed in our work [114]). We find that there are three categories of performance factors in OpenCL: (1) GPU-like programming style—which is, in essence, *bad CPU programming*, (2) fine-grained parallelism—an *intrinsic, fundamental property* of the OpenCL model, and (3) compiler elements.

To quantify the impact of each of these factors, we modify the OpenCL CPU code by removing GPU-specific elements, by tuning the parallelism granularity, and by enabling/disabling specific compiler options. We show that these changes have a significant effect, improving OpenCL performance on multi-core CPUs. Therefore, we argue that OpenCL can be, performance-wise, a good option for multi-core CPU programming among multiple classes of applications and a suitable unified programming model for heterogeneous computing.

The rest of the chapter is organized as follows: Section 2.1 introduces the OpenCL programming model. Section 2.2 specifies our experimental method. In Section 2.3, we present a thorough performance tuning and analysis of each application, leading to a discussion of OpenCL performance impact factors in Section 2.4. Section 2.5 evaluates related studies. In Section 2.6, we summarize this chapter.

2.1 OpenCL Programming Model

OpenCL was proposed by the KHRONOS group in 2008 as an open standard for heterogeneous parallel programming across CPUs, GPUs and other processors [64]. The OpenCL platform consists of a host connected to one or more compute devices. A compute device is divided into multiple compute units (CUs); CUs are further divided into multiple processing elements (PEs); PEs perform the computations (compute kernels).

Figure 2.1 illustrates the OpenCL platform model.

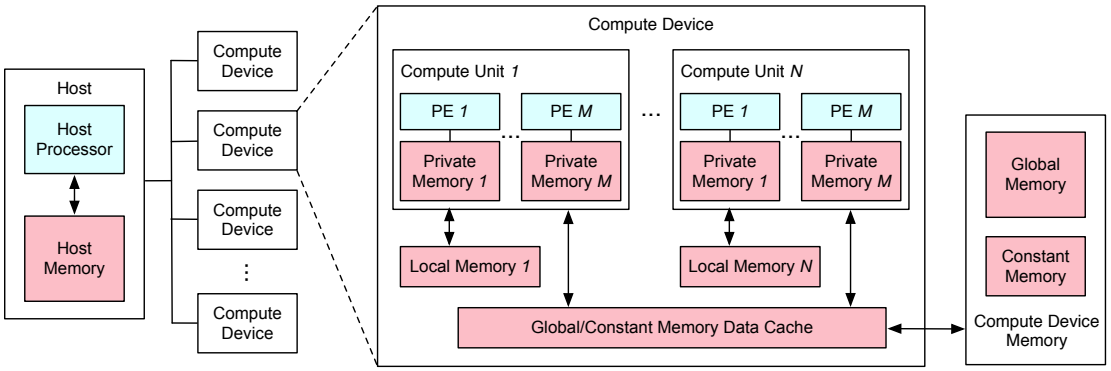


Figure 2.1: The OpenCL platform model and the OpenCL memory model.

An OpenCL program has two parts: the compute kernels that are executed on one or more compute devices, and the host program that runs on the host side. The host program creates a *command-queue* for each device, and enqueues commands to manage the execution of the kernels on the devices. Usually, this management consists of four phases. First, the host defines the context for kernel execution (the initialization phase), including device information collection, kernel program compilation, etc. Then, the host issues memory commands to allocate memory on the devices and transfer input data, if needed, from the host memory to the device memory (the H2D phase). After that, the host coordinates the kernel computation on the devices by using kernel execution commands (the kernel execution phase). When the computation is finished, the host transfers the result data back to the host (the D2H phase). The OpenCL command-queue can be operated in either the in-order execution mode (the enqueued commands are completed in order) or the out-of-order execution mode (the command completion is out of order). The synchronization between commands in a command-queue can be ensured by adding a command-queue barrier or by using the event mechanism.

An instance of a compute kernel is called a *work-item*, and it is executed for each point in the problem space (i.e., parallelization space). As the compute kernel is implemented for each work-item, this ensures fine-grained parallelism. Further, work-items can be organized into *work-groups*, providing a coarser-grained decomposition of the problem space. In OpenCL, the problem space is defined by an N-dimensional index space, called *NDRange*, where N is one, two or three. An NDRange is specified by an N-element tuple with each element indicating the size of each dimension. The work-item geometry in a work-group is specified in the same way and has the same dimensionality. A work-item can be uniquely identified by its global ID in the NDRange or by the combination of its local ID and work-group ID. To parallelize an application, the user maps the problem space to work-items and defines how work-items are grouped into work-groups.

The OpenCL memory model consists of four memory spaces (see Figure 2.1). The global memory is the device memory accessible by all work-items. The constant memory is a region of global memory that is read-only during the execution of a kernel. These two memory spaces can be cached depending on the capabilities of the compute device. The local memory is shared by all work-items within a work-group. It can be implemented in hardware as on-chip scratch-pad memory on the device, or alternatively mapped by the OpenCL runtime/driver onto sections of the global memory. The private memory is exclusively owned by each work-item. OpenCL uses a relaxed consistency memory model. The consistency across work-items in a single work-group can be ensured by a work-group barrier, but there is no explicit mechanism to force memory consistency between different work-groups. Memory updates are guaranteed to be visible to all work-items in all work-groups by using an in-order command-queue and/or at the synchronization points.

2.2 Experimental Method

Our experiments are built to understand the performance behavior of OpenCL on multi-core CPUs and to find out the factors that significantly influence the performance.

2.2.1 Selected Benchmarks

We select our benchmarks from the Rodinia benchmark suite [17]. Rodinia is designed to cover different parallel patterns using the Berkeley Dwarfs [7] as guidelines. Each benchmark contains equivalent implementations in OpenMP, CUDA, and OpenCL. In our work [114], we presented a detailed performance analysis of the OpenMP implementation, and in [113], we presented a wide programmability comparison between OpenMP and OpenCL for eleven different benchmarks. Now, we select five of them (K-means, PathFinder, HotSpot, CFD, and BFS), and study their performance in detail.

Table 2.1: The Rodinia benchmarks used in our experiments.

Benchmark	Dwarf	Domain	Type
K-means	Dense Linear Algebra	Data Mining	Memory-bound
PathFinder	Dynamic Programming	Grid Traversal	Memory-bound
HotSpot	Structured Grid	Physics Simulation	Memory-bound
CFD	Unstructured Grid	Fluid Dynamics	Compute-intensive
BFS	Graph Traversal	Graph Algorithms	Data-dependent

Table 2.1 summarizes the selected benchmarks together with their corresponding dwarfs, application domains, and application types. Our selection covers a diverse range

of parallel patterns and application domains. We also choose different types of applications with respect to their performance bounds: three memory-bound applications (K-means, PathFinder, and Hotspot), and one compute-intensive application (CFD). We note that the ratio between memory-bound and compute-intensive applications is reasonable, because most interesting applications running on the current multi-core processors are constrained by their low arithmetic intensity [131]. BFS is a special case as its performance bound largely depends on the input graph.

2.2.2 Empirical Performance Comparison

We use the OpenMP implementation as a performance estimation of what an application can achieve when using a coarse-grained parallelism model. If the performance difference between the OpenCL and OpenMP implementations is within 10% [113], we consider they have similar performance. If the performance difference is larger than 10%, we tune the OpenCL code step by step to isolate and quantify the performance impact factors. Even if the OpenMP code is not aggressively optimized, as long as it outperforms the OpenCL code, the latter should be further tuned.

Both the OpenCL and OpenMP implementations of an application consist of a sequential component and a parallel component, which are typically executed in some interleaved manner. We measure the wall-clock times (reported in ms) of the parallel component to evaluate OpenCL and OpenMP performance.

In OpenCL, the parallel component can be further divided into four phases: initialization, host to device (H2D) data transfer, kernel execution, and device to host (D2H) data transfer (see Section 2.1). On the one hand, the initialization phase only occurs once in each OpenCL program, and can be reduced by program offline-compilation [3,55]. It further depends on the hardware/software configuration of the machine, and on the execution context, but *not directly* on the application. Moreover, the OpenMP parallel component does not have such initialization overhead. On the other hand, when using OpenCL on CPUs, as the host and the device are the same CPU and share the same main memory, performing real copy in the H2D and D2H phases should be unnecessary. Our work [110] has shown that using zero copy techniques efficiently removes this overhead: no runtime data copies are performed between the host and the device. In other words, H2D and D2H have no impact on the performance comparison, and we can exclude them. Therefore, in our experiments, we compare only the kernel execution phase of each application.

The final comparison is made using the best performance for both the OpenCL and OpenMP implementations.

2.2.3 Experimental Setup

We use three multi-core CPU platforms in our experiments (noted as N8, D6, and MC). The details of each hardware platform are listed in Table 2.2. We run most of experiments and tuning on N8, but we validate our findings on D6 and MC, and comment on the differences.

Table 2.2: The hardware platforms.

Name	Processor	# Cores	# HW threads
N8	2.40GHz Intel Xeon E5620 (hyper-threading)	2× quad-core	16
D6	2.67GHz Intel Xeon X5650 (hyper-threading)	2× six-core	24
MC	2.10GHz AMD Opteron 6172 (Magnycours)	4× twelve-core	48

The compiler we use for OpenMP and the OpenCL host program is GCC 4.4.6, and the compiler options in both are `-O3 -funroll-loops`. Additional tests for OpenMP have been performed using the Intel ICC compiler 12.1. We choose to present the best of the two on an application-by-application basis. For OpenMP, we vary the number of OpenMP threads and choose the best performing one for each application (i.e., the best results achieved by varying the number of threads and the scheduling)[1]. For OpenCL, the kernels are compiled by two different compilers: (1) Intel OpenCL SDK, and (2) AMD Accelerated Parallel Processing (APP) SDK 2.7. Thus, we have two OpenCL versions for each application: an Intel version and an AMD version.

2.3 Performance Evaluation

In this section, we present a thorough performance comparison, tuning, and analysis for each benchmark.

2.3.1 K-means

K-means is a clustering algorithm that uses the mean based data partitioning method [101]. The OpenMP implementation has one parallel section with one parallel `for` loop, replaced by a kernel in the OpenCL implementation. For testing, we use three datasets of 200K, 482K, and 800K objects.

We first vary the work-group size in the OpenCL implementation, and find out that the optimal size is 128 work-items per group in both the Intel and AMD versions (see Figure 2.2). However, except for the first two work-group sizes, the performance variations are within 2%. Therefore, K-means is not sensitive to the work-group size.

[1]More details on the execution times of OpenMP and their variability are given in [114].

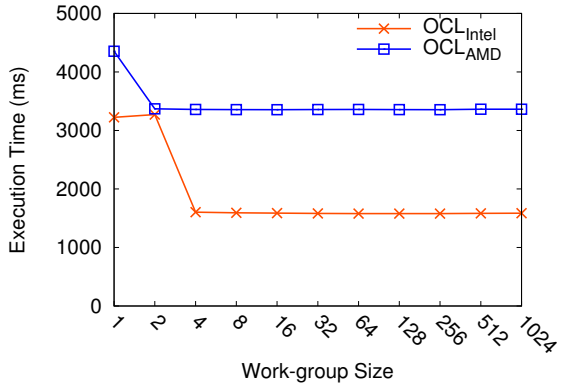


Figure 2.2: K-means OpenCL execution time (ms) with different work-group sizes. The maximum work-group size is limited by the device. For our three hardware platforms, it is 1024 work-items per group.

We also notice that the Intel version performs much better than the AMD version. As the Intel compiler has an implicit vectorization module, it automatically packs multiple work-items executing in parallel in the SIMD unit (the number of work-items is determined by the width of the SIMD unit), thus improving OpenCL performance [105]. In our test, the speedup of using the Intel auto-vectorization is around 2×.

Therefore, we keep the Intel auto-vectorization enabled, set the work-group size in OpenCL to 128, choose the OpenMP best performance (with 48 OpenMP threads), and compare the performance results on different platforms with different compilers in Table 2.3. The arrows indicate the performance of OpenCL vs. OpenMP: better(↗), worse(↘), or similar (↔, difference within 10%).

Table 2.3: K-means execution time (ms) comparison.

Platform	**Dataset**	**OpenMP**	**OpenCL$_{\text{Intel}}$**	**OpenCL$_{\text{AMD}}$**
N8	200K	297.0	308.9 ↔	350.3 ↘
	482K	1600.7	1391.5 ↗	3441.1 ↘
	800K	1403.7	1577.7 ↘	3356.4 ↘
D6	200K	302.7	199.8 ↗	283.3 ↔
	482K	1606.3	851.4 ↗	2107.1 ↘
	800K	1419.1	971.8 ↗	2139.0 ↘
MC	200K	85.6	Fail	428.7 ↘
	482K	505.8	Fail	301.8 ↗
	800K	463.8	Fail	1930.3 ↘

We see that the Intel version outperforms OpenMP in most cases, while the AMD version, due to the lack of auto-vectorization, performs much worse than OpenMP. Note

that the Intel version failed to run on the MC platform (referred as "Fail" in Table 2.3), because the vector extensions SSE4.2 used by Intel OpenCL are not supported by the older MC platform.

Analyzing the K-means implementations, we find that there is a swap kernel in the OpenCL implementation that remaps the data array from the row-major order to the column-major order. The OpenMP implementation has no data layout swapping. Therefore, we measure the K-means OpenCL execution time (the largest dataset) with and without the swap kernel on N8. The performance results are presented in Table 2.4.

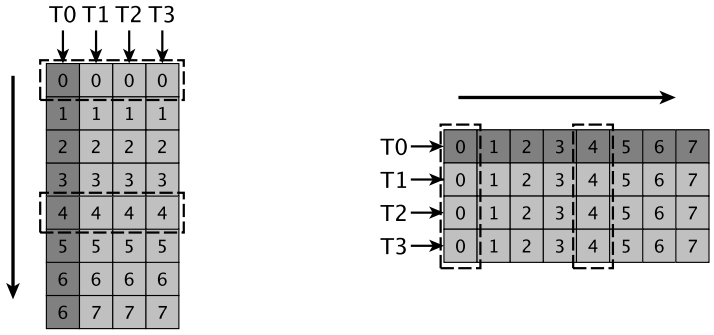
Table 2.4: K-means OpenCL execution time (ms) with and without data layout swapping. The Intel version further has two modes: with auto-vectorization enabled (vec-enabled) and disabled (vec-disabled). The experiments are run on the N8 platform using the largest dataset.

	Intel$_{\text{vec-enabled}}$	**Intel**$_{\text{vec-disabled}}$	**AMD**
With Swapping	1577.7	3330.0	3356.4
Without Swapping	1179.4	614.9	588.3

In the AMD version, the performance without the swap kernel increases significantly, having 5.7× speedup. In the Intel version, the performance speedups with auto-vectorization enabled and disabled are 1.3× and 5.4×, respectively. This performance improvement comes from the different memory access pattern "preference" for CPUs and GPUs. On GPU platforms, array access in the column-major order generates better memory coalescing (neighboring work-items access spatially contiguous data elements). On CPU platforms, such an optimization becomes inefficient. Because a CU and all PEs in the CU are mapped to a CPU hardware core/thread when using OpenCL on CPUs, work-items from the same work-group are queued to run on the same CPU hardware core/thread [3]. With the column-major order (with swapping), each work-item has to access a column of data elements which are not adjacent (the stride is the height of the initial array), resulting in poor data locality both inter-thread and intra-thread. Because K-means is memory-bound [18], memory bandwidth utilization dominates the whole application performance. Therefore, the native row-major order array (without swapping, and similar to OpenMP) delivers better performance when running OpenCL applications on CPUs. This is a known issue as also pointed out by [72] (when porting CPU code onto GPUs), but it has been given a lot less attention on the CPU side before our study.

On the other hand, we see that the Intel auto-vectorization has unexpected performance behavior before and after removing the swap kernel: with swapping, auto-vectorization performs better; removing swapping, it delivers poorer performance. When work-items access their data elements in the column-major order (see Figure 2.3(a)), using auto-vectorization packs four work-items together, and makes them access physically

neighboring data elements per SIMD instruction in the SIMD unit. Therefore, it generates better data locality than the kernel without auto-vectorization. After removing the swap kernel (see Figure 2.3(b)), the native row-major order makes each work-item access a whole row of data elements, generating better caching behavior, while the auto-vectorization leads to non-adjacent data elements processed in the same time.

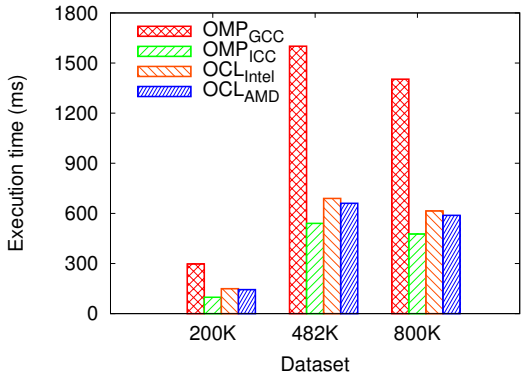


(a) With swapping: Column-major (b) Without swapping: Row-major

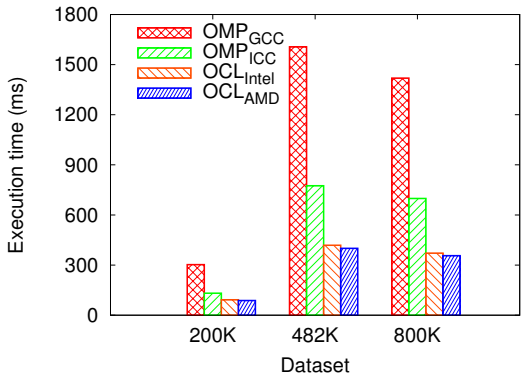
Figure 2.3: The Intel auto-vectorization in K-means with/without swapping and the corresponding memory access patterns. T0–T4 represent work-items. The shaded region together with the numbers represent the memory access sequence in each work-item. The dashed rectangle represents data elements processed together in packed four work-items.

When a kernel processes a 2D dataset element by element (e.g., K-means), using auto-vectorization and using row-major memory access pattern are orthogonal. Auto-vectorization utilizes the SIMD unit to speed up the execution. It also improves cache locality when data elements are accessed in the column-major order compared to the non-vectorized implementation. When the kernel uses the row-major order to preserve cache locality, applying auto-vectorization generates extra overhead. To cover all possibilities, programmers should enable/disable auto-vectorization, switch between column-/row-major order, and choose the best performing combination for their specific case.

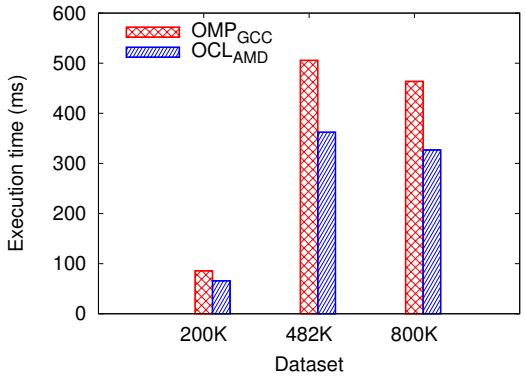
In K-means, the implementation with the row-major order (without swapping) and with auto-vectorization disabled performs the best among the four combinations (see Table 2.4), so we unify all K-means OpenCL experiments (128 work-items per work-group, no swap kernel, no auto-vectorization), and test the performance on all three platforms again. We also test the OpenMP code with the Intel ICC compiler, and we find that the ICC auto-vectorization option improves K-means OpenMP performance (it cannot work on MC, due to the lack of the Intel SSE4.2 instruction support). We show all the results in Figure 2.4. OpenCL (both the Intel and AMD versions) largely outperforms OpenMP with the GCC compiler on all three platforms, and has similar performance (on N8) and better performance (on D6) compared to OpenMP with the ICC compiler.



(a) N8



(b) D6



(c) MC

Figure 2.4: K-means execution time (ms) comparison after performance tuning (128 work-items per work-group, no swap kernel, no auto-vectorization): (a) N8, (b) D6, (c) MC. As the medium dataset needs more iterations to reach the convergence condition, it has larger execution time than the largest dataset. Note that, for readability purposes, the y-axis of the three figures have different scales.

Overall, we see that memory access patterns impact OpenCL CPU performance, especially in memory-bound applications like K-means. We also find that the Intel auto-vectorization can perform differently when using different memory access patterns: adding 2× speedup in the column-major order, and 91% overhead in the row-major order. Thus, programmers need to keep their code in a parameterizable form (e.g., enabling/disabling auto-vectorization, switching between column-/row-major order), and tune the code to obtain the best performance.

2.3.2 PathFinder

PathFinder uses dynamic programming to find paths on a 2D grid [86]. For each data element in the bottom row of the grid, PathFinder finds its shortest path to its corresponding position in the top row by choosing the smallest accumulated weight. The application iterates row by row, comparing and updating the accumulated weights for all data elements of a row in parallel (1D parallelization). Thus, a parallel `for` is implemented in OpenMP and replaced by a kernel in OpenCL. For PathFinder, we use three grids with 100000, 200000, and 400000 (100K, 200K, and 400K) data elements per row.

This is a typical application that uses iterative stencil loops (ISL) [75]. We find that the original Rodinia OpenCL implementation uses OpenCL local memory to apply ghost zone optimization [86], improving PathFinder performance on GPUs. However, this optimization does not work well on CPUs (in our findings [113], OpenMP is 5 times faster than the original Intel version, and the AMD version produces unstable code for all platforms).

Because CPUs do not have a special hardware resource designed as local memory, all memory buffers in local memory are mapped onto sections of global memory and cached by hardware [54]. Explicit caching through local memory introduces additional write operations to copy data from global memory to local memory, and work-group barriers for data consistency [3] (we found this by inspecting the assembly code). Thus, we re-implement a naive kernel for PathFinder, using the same approach as the OpenMP implementation and using only global memory.

In this naive kernel, one work-item processes one data element of one row: it compares its accumulated value with its left and right neighbors, and adds the minimum one to the same column position of the next row. The kernel is invoked iteratively for a number of times equal with the number of rows. Because every work-item on the borders only has one neighbor, we must use two work-item ID dependent branches (conditional branches depending on the work-item ID) to make the border work-items process correctly.

As the Intel compiler can now auto-vectorize work-item ID dependent branches, we first measure the kernel execution time with auto-vectorization enabled and disabled. Our test shows that the kernel with auto-vectorization generates 26%-71% overhead (more

details about the cause of the overhead are given in Section 2.3.3). Thus, we decide to disable auto-vectorization in the Intel version. We further vary the work-group size, and find that the performance improves with the increase of the work-group size (see Figure 2.5). This is mainly because a smaller number of work-groups (a larger work-group size) reduces the CPU scheduling overhead. Therefore, we use 1024 work-items per group in our experiments.

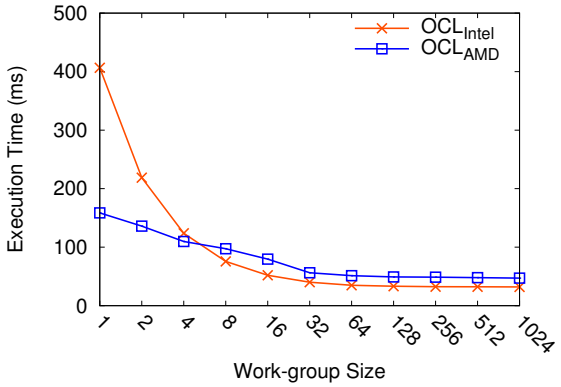


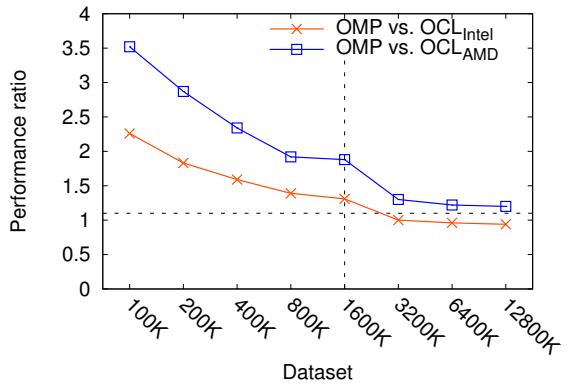
Figure 2.5: PathFinder OpenCL execution time (ms) with different work-group sizes.

We present the performance comparison between OpenMP and OpenCL for the three platforms in Table 2.5. OpenCL cannot compete with OpenMP for all platforms and datasets: the performance is around 2 times worse in the Intel version and 4 times worse in the AMD version. Therefore, we continue increasing the dataset size to 12800K, and compare the performance again on the N8 platform. In Figure 2.6(a), we find that the performance ratio of OpenMP to OpenCL decreases, for both the Intel and AMD versions, with the increase of the dataset size (in other words, OpenCL gets better and/or OpenMP gets worse). For the largest dataset, the ratio already drops to around 1.1. Why is this happening?

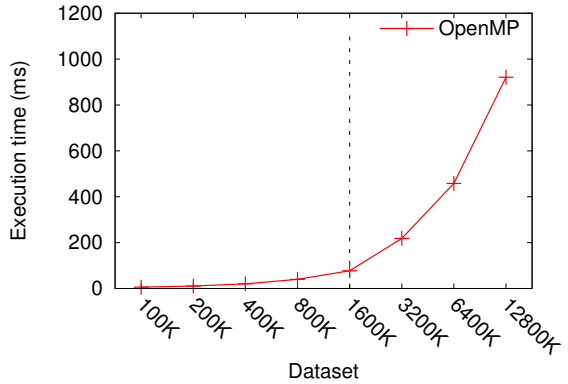
Table 2.5: PathFinder execution time (ms) comparison.

Platform	**Dataset**	**OpenMP**	**OpenCL$_{\text{Intel}}$**	**OpenCL$_{\text{AMD}}$**
N8	100000	5.7	12.9 ↘	20.0 ↘
	200000	10.7	19.5 ↘	30.6 ↘
	400000	20.3	32.2 ↘	47.5 ↘
D6	100000	3.9	10.7 ↘	15.4 ↘
	200000	6.9	15.9 ↘	24.6 ↘
	400000	12.9	26.3 ↘	34.8 ↘
MC	100000	6.9	Fail	58.9 ↘
	200000	14.3	Fail	68.7 ↘
	400000	28.6	Fail	89.4 ↘

As PathFinder is not a compute-intensive application, the performance is bound by the memory access. In OpenMP (the best performance with 8 threads), one thread, running on one hardware core, processes contiguous data elements in memory. Thus, it makes use of the cache to improve application performance. In Figure 2.6(b), we notice a turning point at the 1600K dataset, where the OpenMP performance is reduced largely. We believe this is because larger datasets (>1600K) exceed the capacity of the N8 cache, which leads to the sudden OpenMP performance degradation. For the same reason, the performance ratio shown in Figure 2.6(a) also drops fast from this point on, becoming approximately 1.1 at last. We further change the array index from the row-major order to the column-major order (opposite to the previous experiments, thus unfriendly for the CPU cache). The OpenMP performance becomes 2-6 times worse, confirming our hypothesis that caching is the main reason for OpenMP to drastically outperform OpenCL in PathFinder.



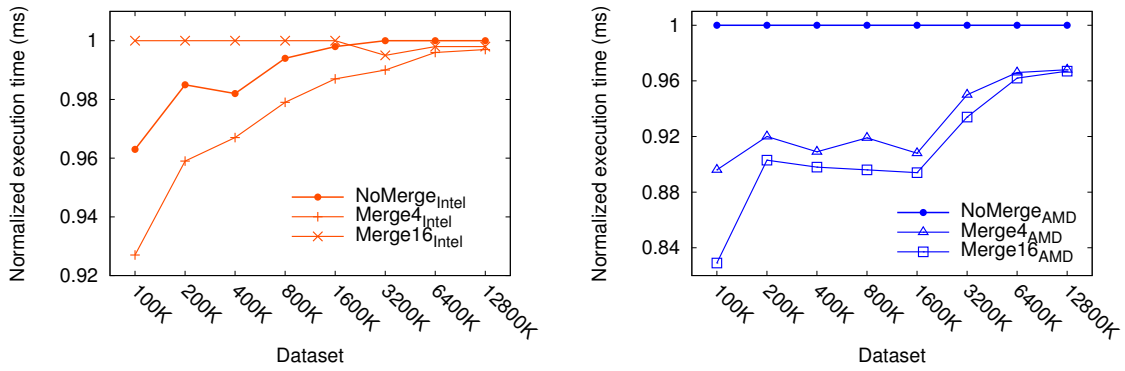
(a) Performance ratio



(b) OpenMP performance

Figure 2.6: PathFinder performance with increased dataset size: (a) the performance ratio of OpenMP to OpenCL, (b) the OpenMP performance measured in execution time (ms).

We also notice that for larger datasets (>1600K), the Intel version already performs slightly better than OpenMP, which means that when parallelizing the problem space in one dimension (e.g., PathFinder), OpenCL is suitable for large-scale data parallelism, while it has deficiencies for smaller datasets. We note that in OpenCL, each work-item of a work-group, queueing to run on the same hardware core/thread, only processes one data element. This fine-grained parallelism restricts cache utilization among work-items on CPUs. Therefore, we use an alternative way (we call it the *MergeN* optimization) to explicitly increase the workload for each work-item: we make each work-item process N adjacent data elements; we use a loop to duplicate the statements and unroll the loop using the OpenCL `unroll` pragma.



(a) The OpenCL Intel version

(b) The OpenCL AMD version

Figure 2.7: PathFinder OpenCL normalized execution time (ms) with the MergeN optimization, N = 4, 16. "NoMerge" means without the MergeN optimization. Normalized execution time for each dataset = $log_{10}(x)/log_{10}(max)$, x = the execution time of Merge4, Merge16, and NoMerge, max = the maximum execution time among Merge4, Merge16, and NoMerge. Note that, for readability purposes, the y-axis of the two figures have different scales.

Figure 2.7 shows the performance of applying the MergeN optimization (N = 4, 16). We see that MergeN behaves differently in the two OpenCL versions. When using the AMD compiler, the performance is largely improved by 26% after merging 4 work-items, and further improved by 7% by applying Merge16. In the Intel version, Merge4 improves performance by 5%, while Merge16 performs even worse than NoMerge, adding 2% overhead. We find that Merge16 generates register spills and reloads, which slows down its execution, while NoMerge and Merge4 do not have this problem. In general, when we increase the workload per work-item, we improve the data locality within each work-item and generates better caching. Thus, we conclude that MergeN is a good option for improving caching performance, but "abusing" it may lead to register spills and, consequently worse performance. In addition, although the Intel version has less efficient

merging, its Merge4 already generates equivalent performance with AMD's Merge16, which shows that the Intel compiler has more efficient cache utilization by default.

In the final comparison, we use Merge4 and Merge16 for the Intel version and the AMD version, respectively. The final results for all three platforms are presented in Figure 2.8. We see that OpenCL on all three platforms achieves similar or better performance for datasets larger than 1600K. We note that the OpenMP results presented in Figure 2.8 are obtained with GCC, which generates better performance than ICC.

To summarize, our PathFinder implementation shows the performance impact of (destroying) data locality when using 1D parallelization: in OpenCL, the inherent fine-grained work-items can have a negative impact on the original locality on CPUs, while OpenMP's coarse-grained parallelization preserves it. A solution to solve this penalty is to increase the parallelism granularity in OpenCL. As a side effect, the original PathFinder implementation in Rodinia has proved that the use of local memory can be a disadvantage in OpenCL, as this memory is not mapped on a special, faster physical memory on CPUs.

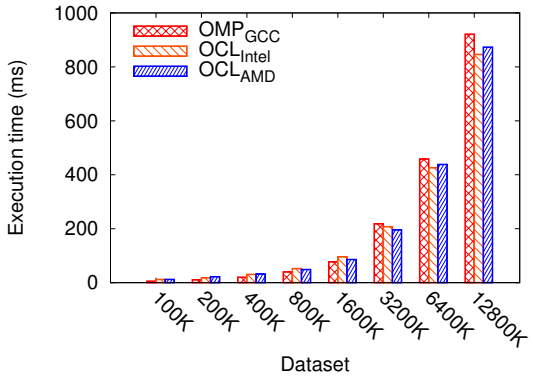
2.3.3 HotSpot

HotSpot is a 2D transient thermal modeling kernel [50], which computes the final state of a grid of cells when given the initial conditions (temperature and power dissipation per cell). The application iteratively updates the temperature values of all cells in parallel, and usually stops after a given number of iterations. For HotSpot, we have three datasets, for grids of 64×64 (4K), 512×512 (256K), and 1024×1024 (1M) cells.

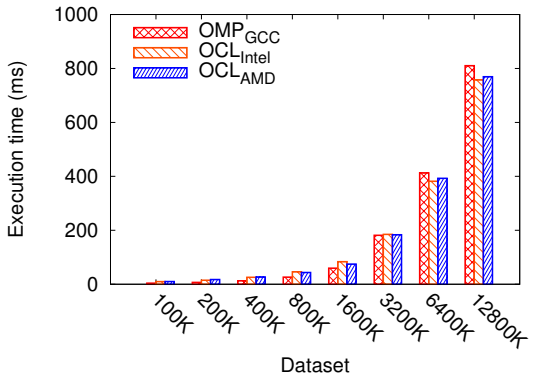
HotSpot is another ISL application, and the original Rodinia implementation also uses OpenCL local memory and ghost zone optimization to reduce synchronizations and communications by redundant computation [86]. As the OpenMP implementation does not have such optimization and using local memory on CPUs can introduce overhead (see Section 2.3.2), we re-implement a naive OpenCL HotSpot similar to that implemented in OpenMP: the two OpenMP parallel `for` loops, for the grid computation and the grid update, are replaced by two OpenCL kernels.

We note that the Intel implicit vectorization module auto-vectorizes the kernels. Table 2.6 shows the performance differences with auto-vectorization enabled and disabled on N8. As the computation kernel uses nine work-item ID dependent branches to process border data elements (e.g., grid corners and edges), the possibility of four work-items executing different branches in the SIMD unit (when auto-vectorization is enabled) is relatively high for the border regions. The compiler has to mask SIMD lanes to ensure that work-items correctly execute their corresponding `if` or `else` branches, resulting in significant performance hit [52].

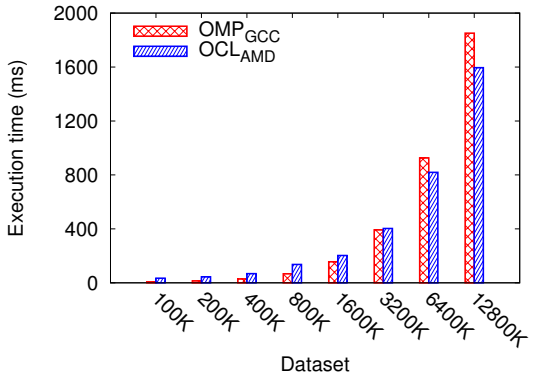
To verify the impact of work-item ID dependent branches, we artificially remove them (i.e., we do not treat border regions in a special way) at the expense of functional correct-



(a) N8



(b) D6



(c) MC

Figure 2.8: PathFinder execution time (ms) comparison after using the MergeN optimization in OpenCL: (a) N8, (b) D6, (c) MC. Note that, for readability purposes, the y-axis of the three figures have different scales.

Table 2.6: Execution time (ms) of the OpenCL Intel version with auto-vectorization enabled (vec-enabled) and disabled (vec-disabled). We first measure the execution time of the computation kernel with branches. Then we remove the branches and measure the execution time again.

HotSpot with branches	**vec-enabled**	**vec-disabled**
64 × 64	186.0	78.3
512 × 512	8507.0	752.4
1K × 1K	33439.8	2667.9
HotSpot without branches	**vec-enabled**	**vec-disabled**
64 × 64	43.8	51.9
512 × 512	260.8	674.2
1K × 1K	segmentation fault	

ness (which could be preserved by data padding). We measure the performance with auto-vectorization enabled and disabled again. Without branches, enabling auto-vectorization largely reduces the execution time (see Table 2.6). Therefore, the performance of Intel's auto-vectorization is affected by work-item ID dependent branches: auto-vectorization delivers performance improvement for kernels with no work-item ID dependent branches, but can lead to performance degradation if such branches are present.

We further build three kernels that contain one main branch with: (1) four corner branches, (2) two corner branches and two edge branches, and (3) four edge branches, respectively. We find that the number of branches has no direct impact on performance—i.e., the third kernel is two times slower than the first kernel, and different choices of corners and edges in the second kernel have up to nine times performance gap. Instead, it is the combination of the number of branches and the probability of taking each of the `if` or `else` branch (which is related to the data) that make a difference. Thus, programmers should remove/limit work-item ID dependent branches in the kernel, or simply disable the implicit vectorization module, as in this case.

In the next step, we consider merging the two kernels into one to reduce the number of kernel invocations. This requires a work-group barrier between them to ensure correct ordering of the accesses to global memory within a work-group. In the Intel version, merging reduces the execution time by around 14%, but in the AMD version, it leads to worse performance. Our results in Table 2.7 show that the barrier operation is considerably heavier in the AMD version, where the overhead of the barrier far exceeds the benefit of merging. Without the barrier, merging would manifest its advantage, reducing the total execution time by 23%. We use the similar barrier directive in OpenMP to merge the two parallel sections, but we get an even larger performance penalty.

We also systematically tune the work-group size in OpenCL. Figure 2.9 shows that

Table 2.7: Barrier overhead in the OpenCL AMD version reflected by the execution time (ms). The second column represents the original two-kernel implementation. The third column represents the one-kernel implementation by adding a barrier. The fourth column represents the one-kernel implementation with commenting the barrier.

	Two kernels	**One kernel with a barrier**	**One kernel with the barrier commented**
64 × 64	137.4	146.4	91.8
512 × 512	1158.6	4577.6	1127.3
1K × 1K	3895.4	18947.6	3573.2

the HotSpot performance is improved with the increase of the work-group size, making 32×32 the optimal one. According to all these findings, we unify our experiments (two-kernel implementation, Intel auto-vectorization disabled, and 32×32 work-group size), and present the execution time comparison between OpenMP and OpenCL in Table 2.8. When the dataset size is small enough (i.e., 64×64) to fit into the cache, OpenMP performs better than OpenCL. For the two larger datasets, the Intel version outperforms OpenMP by 18%/35%, and the AMD version has similar performance compared to OpenMP. The exception is MC, mainly because OpenMP is more friendly to its NUMA architecture [13], but the performance gap between OpenMP and OpenCL is also decreased with the increase of the dataset size. Besides, we note that for OpenMP, the GCC compiler outperforms again the ICC compiler for all datasets and platforms.

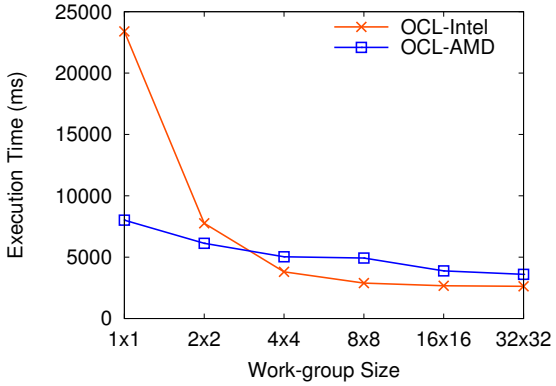


Figure 2.9: HotSpot OpenCL execution time (ms) with different work-group sizes.

Why does OpenCL have better or equivalent performance with larger datasets? We notice that the two programming models have different data distributions. In OpenCL, the whole grid of data is distributed tile by tile, in a cyclic way. OpenMP uses block distribution, with one thread processing a stripe of the whole grid (the height of the stripe is determined by the number of grid rows and the number of OpenMP threads). We vary

Table 2.8: HotSpot execution time (ms) comparison.

Platform	Dataset	OpenMP	$\text{OpenCL}_{\text{Intel}}$	$\text{OpenCL}_{\text{AMD}}$
N8	64 × 64	23.8	110.2 ↘	100.0 ↘
	512 × 512	901.2	756.7 ↗	1040.6 ↔
	1K × 1K	3339.3	2625.2 ↗	3597.9 ↔
D6	64 × 64	19.5	100.0 ↘	88.6 ↘
	512 × 512	589.6	500.6 ↗	702.8 ↔
	1K × 1K	2361.7	1649.9 ↗	2205.4 ↔
MC	64 × 64	27.3	Fail	267.2 ↘
	512 × 512	301.6	Fail	1644.3 ↘
	1K × 1K	2057.5	Fail	3045.5 ↘

the data distribution in both OpenMP and OpenCL to evaluate the performance impact of the distribution type:

(a) Change OpenMP data distribution from stripe-based to tile-based.

(b) Change OpenCL data distribution from tile-based to stripe-based, with each work-group processing one $1 \times$ *grid width* stripe (one row, the height of the stripe is limited by the maximum work-group size).

(c) Use the same distribution (the same as (b)) in OpenMP by setting the scheduling parameter to *(static, 1)*, such that each row is also processed by one OpenMP thread.

We run the test with the largest dataset on N8. The execution time comparison in Table 2.9 shows that (1) the tile-based distribution outperforms the stripe-based distribution in OpenCL, and (2) OpenMP delivers better performance using the stripe-based distribution. When parallelizing 2D applications (e.g., HotSpot), OpenCL's tile-based data distribution performs better (in the Intel version) or at par (in the AMD version) as OpenMP's default stripe-based distribution, because OpenCL's tile-based distribution generates better scheduling and caching for 2D parallelism on CPU platforms.

In summary, we have shown that the Intel and AMD compilers have different OpenCL implementations, leading to different performance behavior: Intel's implicit vectorization module can positively impact OpenCL performance, but its combination with divergent branches can lead to performance penalties. AMD's barrier is such a heavy operation that programmers must limit its use as much as possible. Furthermore, we also point out that OpenCL and OpenMP have their "preferred" data distribution type and granularity. Finally, we note that OpenCL has more flexibility and performance benefits in implementing tile-based data distribution, an operation we found very difficult to perform in OpenMP (a research topic in itself [32]).

Table 2.9: The execution time (ms) comparison of the tile-based and stripe-based data distributions in OpenMP and OpenCL.

OpenMP 32×32 tile-based	**OpenCL 32×32 tile-based**	
	Intel	**AMD**
4462.9	2625.2	3597.9
OpenMP 1×1024 stripe-based	**OpenCL 1×1024 stripe-based**	
	Intel	**AMD**
3374.4	3675.2	4112.2

2.3.4 CFD

The CFD Solver is an unstructured grid finite volume solver for the 3D Euler equations for inviscid, compressible flow [20]. For CFD, we choose a parallelism-rich version (i.e., the single precision version with redundant flux computation), with reduced total memory accesses and high arithmetic intensity. The OpenMP version uses four parallel `for` loops for the initialization and for the main computation. Each parallel loop is replaced by a kernel in the OpenCL version. We use three datasets: fvcorr.domn.097K, fvcorr.domn.197K, and missile.domn.0.2M.

First, we find that CFD is not sensitive to the work-group size. The execution time difference is within 1% when varying the work-group size. As CFD already has an application-related work-group size coded in the benchmark (192 work-items per group), we use this default work-group size in OpenCL. Checking the OpenCL code, we notice that although there is no apparent swap kernel like in K-means, CFD also attempts to improve GPU memory coalescing by using the column-major data accesses when reading input datasets and accessing memory buffers. We also find that one of the computation kernels which iterates 24000 times, has a 3-way data dependent branch for each iteration. Therefore, we disable the Intel implicit vectorization model (more details can be found in our work [110]), change the array index back to the row-major order (see Section 2.3.1), and compare the OpenCL performance with the best OpenMP performance in Table 2.10.

We see that the OpenCL AMD version performs at par or better than OpenMP on N8 and D6 (MC is again lagging behind), while the Intel version is worse for all cases. Table 2.11 shows the performance speedup after index changing from column-major to row-major for all three datasets on the N8 platform. We see that this change improves performance only slightly (within 10%). This small impact on performance is due to the fact that CFD is a compute-intensive application (it performs a lot of floating-point operations, which dominate the execution time). Moreover, it uses a redundant computation scheme to reduce memory accesses, making the computation even more intensive.

Thus, we focus on the computation part, and utilize the OpenCL `-cl-fast-relaxed-math` flag on the row-major OpenCL implementation.

Table 2.10: CFD execution time (ms) comparison.

Platform	Dataset	OpenMP	OpenCL$_{\text{Intel}}$	OpenCL$_{\text{AMD}}$
N8	fvcorr.domn.097K	22354.5	34086.2 ↘	23742.5 ↔
	fvcorr.domn.197K	45065.9	67614.8 ↘	48338.6 ↔
	missile.domn.0.2M	62589.6	139214.0 ↘	49907.9 ↗
D6	fvcorr.domn.097K	19572.9	21974.2 ↘	15481.3 ↗
	fvcorr.domn.197K	38094.2	43332.8 ↘	31458.9 ↗
	missile.domn.0.2M	49943.3	86678.0 ↘	33099.3 ↗
MC	fvcorr.domn.097K	9353.6	Fail	18897.8 ↘
	fvcorr.domn.197K	16037.1	Fail	36986.1 ↘
	missile.domn.0.2M	46766.4	Fail	42610.8 ↔

Table 2.11: CFD OpenCL performance speedup after index changing from column-major to row-major.

Speedup	OpenCL$_{\text{Intel}}$	OpenCL$_{\text{AMD}}$
fvcorr.domn.097K	1.03	1.01
fvcorr.domn.193K	1.10	1.09
missile.domn.0.2M	1.04	1.01

This flag contains three floating-point optimizations: (a) allow $a * b + c$ to be replaced by a `mad` operation with reduced accuracy; (b) ignore the signedness of zero; (c) allow optimizations for floating-point arithmetic that assume that arguments and results are not infinite [64]. Turning this flag on makes the compiler trade application precision for a higher performance and a potentially less accurate result. We further compare the CFD results with and without `-cl-fast-relaxed-math` flag, and find that the accuracy is sufficient for this benchmark. Figure 2.10 shows the CFD performance improvement after applying this flag on the N8 platform.

We see that for the Intel version, the performance of smaller datasets increases by 26%, while that of the largest dataset increases by 64%. The difference in performance improvement is mainly due to the fact that the datasets are from different fluid flows [110]. On the other hand, the AMD compiler does not make any visible improvement. After setting this flag on, the Intel version performs the same as the AMD version. We note that we also apply this flag in other benchmarks. As these benchmarks are not compute-intensive, they do not show significant performance improvement.

Combining these observations, we believe that `-cl-fast-relaxed-math` is an effective OpenCL optimization for compute-intensive applications like CFD. We also guess that Intel and AMD have different specific implementations of this optimization: AMD partially enables floating-point arithmetic related optimizations by default, while Intel requires this flag to be set explicitly.

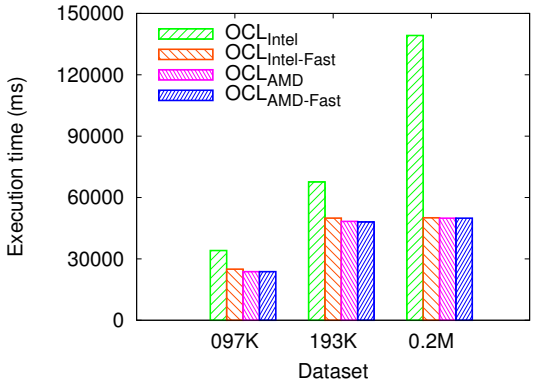


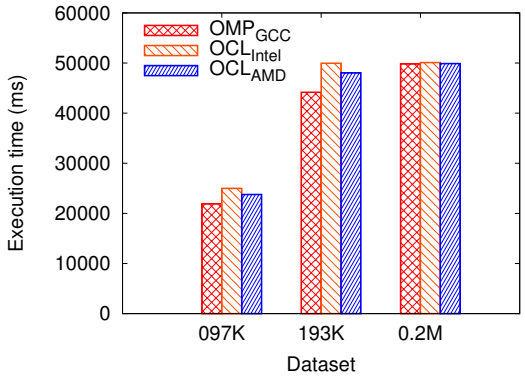
Figure 2.10: CFD OpenCL execution time (ms) with and without *-cl-fast-relaxed-math* on N8. "Fast" represents with this flag on.

We also test the equivalent "fast math" optimization on the OpenMP code, and present the optimized OpenMP and OpenCL performance in Figure 2.11. In most cases, OpenCL has equivalent performance with OpenMP, with the exception on MC for the smaller datasets (the original OpenMP cases already perform much better than the OpenCL ones). The "fast math" speedup comparison in Table 2.12 further shows that OpenCL allows for more aggressive arithmetic optimizations than OpenMP. Compared to the OpenMP GCC version, our additional tests with the ICC compiler do not show any visible improvement.

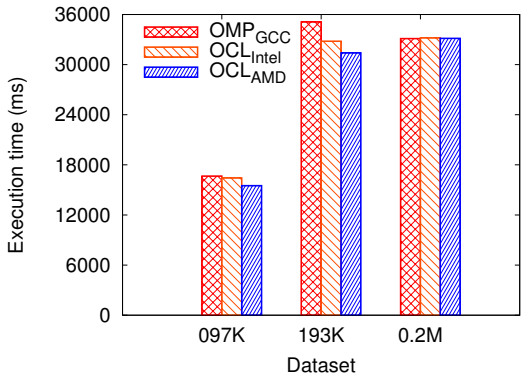
Table 2.12: The performance speedup after applying the "fast math" optimization in OpenMP, the OpenCL Intel version, and the OpenCL AMD version, respectively. For each dataset, we show the minimum and maximum speedup among the three platforms. AMD does not show visible speedup, because AMD partially enables "fast math" by default.

Speedup	**OpenMP**		**OpenCL$_{\text{Intel}}$**		**OpenCL$_{\text{AMD}}$**	
	Min	**Max**	**Min**	**Max**	**Min**	**Max**
fvcorr.domn.097K	1.02	1.21	1.34	1.36	1.00	1.00
fvcorr.domn.193K	1.02	1.08	1.32	1.35	1.00	1.04
missile.domn.0.2M	1.17	1.51	2.61	2.78	1.00	1.02

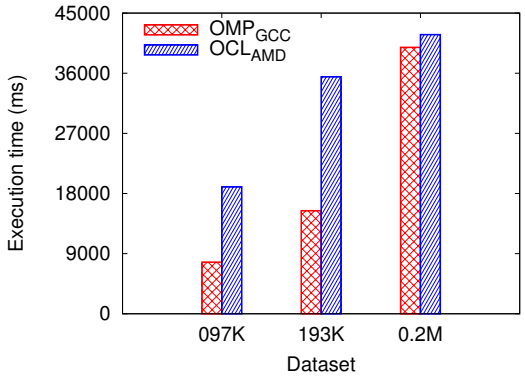
In summary, for compute-intensive applications, the optimizations in the computation part (e.g., the "fast math" optimization) brings in more performance improvement than the optimizations in the memory access part (e.g., using the row-major memory accesses). After tuning, OpenCL and OpenMP deliver similar performance in the CFD application, but the OpenCL "fast math" optimization is more efficient.



(a) N8



(b) D6



(c) MC

Figure 2.11: CFD execution time (ms) comparison after applying the “fast math” optimization: (a) N8, (b) D6, (c) MC. Note that, for readability purposes, the y-axis of the three figures have different scales.

2.3.5 BFS

BFS (Breadth-First Search) is a fundamental graph traversal algorithm. We test the Rodinia parallel implementation [42] on three graph datasets, consisting of 4K, 64K, and 1M nodes. There are two parallel `for` loops executed multiple times in OpenMP for graph traversal and state update, which are implemented as two kernels in OpenCL.

First, we see that the OpenCL kernels contain data dependent branches. According to our experiments, we disable the Intel implicit vectorization module, as the execution time without auto-vectorization is 5%–12% smaller than that with it (similar finding is presented in Section 2.3.4). We also vary the work-group size and set the best performing one (which is 1024 work-items per group). Table 2.13 presents the performance comparison of OpenMP and OpenCL. OpenMP is compiled with the GCC compiler, because our test shows that the GCC compiler generates slightly better performance than the ICC compiler.

Table 2.13: BFS Execution time (ms) comparison.

Platform	Dataset	OpenMP	$\text{OpenCL}_{\text{Intel}}$	$\text{OpenCL}_{\text{AMD}}$
N8	4K	0.504	1.492 ↘	2.256 ↘
	64K	2.360	3.323 ↘	3.887 ↘
	1M	35.854	33.859 ↔	41.619 ↘
D6	4K	0.434	1.390 ↘	1.744 ↘
	64K	2.104	2.850 ↘	3.747 ↘
	1M	29.667	28.724 ↔	36.952 ↘
MC	4K	0.577	Fail	3.002 ↘
	64K	6.251	Fail	17.081 ↘
	1M	89.886	Fail	148.132 ↘

OpenMP outperforms OpenCL in most cases and the largest difference comes from the AMD version on MC. On all three platforms, the performance difference decreases with the increase of the dataset size. In BFS, traversing a graph causes random jumps in memory and low cache efficiency [42]. In addition, since the number of target nodes that a node spawns can vary widely, the workloads among threads are imbalanced. Therefore, the graph attributes (e.g., the number of nodes, the number of edges, sparse/dense graph, etc.) can largely affect the graph traversing performance.

We perform two comparative experiments to show how different graphs influence the BFS performance. In the first experiment, we generate a set of five synthetic graphs, each of which has 10M nodes and different average degrees: 4, 6, 8, 10, and 12. The degree of a node is the number of direct neighbors it has. We run BFS on this set of graphs on N8, and report the best performance for OpenMP and OpenCL in Figure 2.12. First, after increasing the number of nodes to 10M, both OpenCL versions perform better

than OpenMP (with one exception: the AMD OpenCL version and OpenMP perform the same on the 10M-4 graph). Second, with increasing the average degree, the performance differences are gradually enlarged in favor of OpenCL.

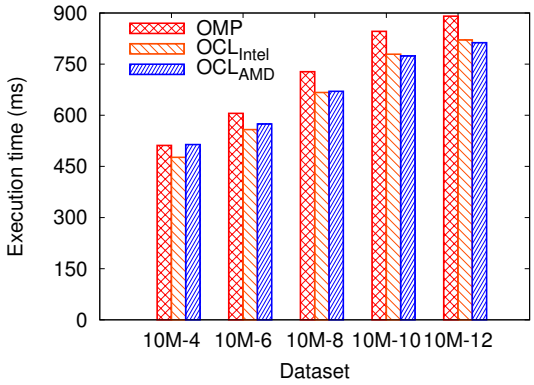


Figure 2.12: BFS execution time (ms) of traversing graphs with different average degrees. We use five synthetic graphs. Each graph has 10M nodes with different average degrees: 4, 6, 8, 10, and 12.

In the second experiment, we create two graphs of 1M nodes with special shapes: a "chain" graph and a "star" graph. Table 2.14 shows the performance comparison. We see that the graph shape significantly affects the BFS performance. In the "chain" graph, nodes are processed one by one, so parallel execution actually degrades to sequential execution. OpenMP generates much less execution overhead and/or more efficient caching in this case: it performs one and two orders of magnitude better than the Intel version and the AMD version, respectively. In the "star" graph, all the target nodes are independently connected to the single source node, and should be processed in parallel in the second iteration. OpenCL's fine-grained parallelism matches this shape of graph well, leading to better (the Intel version) or similar (the AMD version) performance compared to OpenMP.

Table 2.14: BFS execution time (ms) of traversing special shaped graphs: a "chain" graph and a "star" graph. Each graph has 1M nodes.

Shape	**OpenMP**	**OpenCL**$_{\text{Intel}}$	**OpenCL**$_{\text{AMD}}$
1M-chain	229265.3	781483.0	1331470.0
1M-star	18.5	16.0	18.7

To sum up, as graphs can be widely different, and the BFS performance is so sensitive to the graph itself, there is no clear indication that one programming model will always outperform the other (it seems that the coarse-grained OpenMP is suitable for graphs with lower degrees and larger depths, while the fine-grained OpenCL matches the graphs with higher degrees and smaller depths). Due to BFS's graph-dependent performance

behavior, generalizing an optimization rule for improving its performance is impossible without characterizing the graph dataset, and this is outside the scope of this work.

2.4 Performance Impact Factors

We have seen so far that when using OpenCL on multi-core CPUs, the resulting performance is a combined effect of multiple performance impact factors. Table 2.15 summarizes our experimental findings, including the impact factors, their effects (positive or negative) on OpenCL performance, in which application we have found them, and the impact factor category.

Table 2.15: OpenCL Performance Impact Factors.

Factor	Effect	Application	Category
Auto vectorization	Positive	K-means	OpenCL Compilers
Fast math	Positive	CFD	OpenCL Compilers
Flexible data distribution	Positive	HotSpot	Parallelism granularity
Fine-grained parallelism	Positve	HotSpot, BFS	Parallelism granularity
Memory access pattern	Negative	K-means, CFD	GPU programming style
Local memory	Negative	PathFinder, HotSpot	GPU programming style
Auto vectorization	Negative	PathFinder, HotSpot, CFD, BFS	OpenCL Compilers
Barrier	Negative	HotSpot	OpenCL Compilers
Fine-grained parallelism	Negative	PathFinder, BFS	Parallelism granularity

Incorrect GPU programming style appears because most of the OpenCL implementations in Rodinia (and elsewhere) are directly translated from their CUDA versions, and GPU-specific solutions are directly inherited. For the studied benchmarks, the use of the column-major (instead of row-major) data access pattern and the use of local memory have a negative impact on performance when applied on CPUs. Therefore, these hardware-dependent performance factors have to be removed when porting GPU OpenCL code to CPUs.

Parallelism granularity differences are intuitive at the programming model level: OpenMP uses coarse-grained parallelism by splitting loop execution on multiple threads, while OpenCL uses fine-grained parallelism by implementing the computation for each work-item and combining multiple work-items in work-groups. OpenCL's fine-grained parallelism can be both an advantage and a disadvantage in different application scenarios when compared with OpenMP. In our experiments, it leads to poorer data locality (due to the OpenCL platform mapping on CPUs) in 1D PathFinder, while it also generates a more flexible data distribution in 2D HotSpot. As the parallelism granularity has application-dependent performance impact, systematic tuning should be applied to find the optimal

parallelism granularity, which can be easily expressed in OpenCL, but it requires more work in OpenMP.

OpenCL compilers are still under development, thus not fully mature. In our work, we have continuously updated the compilers up to their latest versions, and our experience with compiler evolution is mixed. On the one side, the latest compilers improve OpenCL kernel performance by 20%-30%, and provide more aggressive optimizations in floating-point operations; on the other side, the heavy barrier operation and the host-device data transfers are not yet improved, challenging programmers to limit barrier operations and utilize zero copy techniques. Furthermore, optimizations that should be promising in terms of performance can back-fire. Take the Intel implicit vectorization module for example: the previous compiler can only support data dependent branches, while the latest compiler can also vectorize work-item ID dependent branches. However, with auto-vectorization possibility increasing, more kernels with data or work-item ID dependent branches may suffer from the negative impact of auto-vectorization. In addition, the Intel compiler cannot make use of older platforms (e.g., AMD's Magnycours), given its use of the latest SSE instructions. Generally, it seems that OpenCL compilers' maturity and optimizations should not (yet) be taken for granted. In this context, our findings can be used to check, improve, and correct future compilers.

In our experiments, we have shown that OpenCL code (K-means, PathFinder, CFD) can be tuned to match OpenMP's "regular" performance. Part of this tuning is needed because users are negligent in porting OpenCL code from GPUs to CPUs, part because OpenCL is not properly mapped on CPUs, and part because compilers are not yet mature. Furthermore, we have also proved that the strongest point of OpenCL remains its ability to express fine-grained parallelism (HotSpot, BFS), enabling, for example, flexible data distribution and implicit vectorization at the compiler level.

Because OpenCL can achieve matching performance on CPUs (either after performance tuning, or due to its fine-grained parallelism approach), it is a good option for CPU programming, and eventually for heterogeneous computing. To make good use of CPUs and GPUs, OpenCL needs code specialization to best utilize different hardware architectures, but we point out that the code specialization comes in the form of parameter tuning (e.g., changing parallelism granularity, switching between row/column major order, enabling/disabling zero copy, enabling/disabling auto-vectorization, etc.), and does not alter the application structure and parallelization approach. Therefore, we consider this as an important argument in favor of using OpenCL as a unified programming model for heterogeneous platforms.

2.5 Related Work

In this section, we discuss the related studies of using OpenCL on CPUs. There are multiple studies on OpenMP and other CPU specific programming models [62, 92], and also some relevant work on studying OpenCL on GPUs [29, 61]. However, very little research is available on evaluating OpenCL performance on multi-core CPUs.

Membarth et al. [85] evaluated five parallel programming models (OpenMP, Cilk++, TBB, RapidMind, and OpenCL) on multi-core CPUs. They compared the five programming models from different aspects, such as performance and usability. According to their results, OpenMP and TBB obtain the best performance, while OpenCL is outstanding only for large datasets. In their work, only one application (compute-intensive) from medical imaging is used. Our work covers a diverse range of applications, leading to a more comprehensive understanding of the OpenCL performance on CPUs.

Ali et al. [2] also presented a comparative study of three programming models: OpenMP, TBB, and OpenCL. They chose five synthetic applications, and run experiments on a single Intel CPU. They mainly focused on the impact of compiler optimization options and the scalability of the programming models. Their results show that OpenCL yields similar performance when compared with the other two, but OpenMP is still the best among the three programming models. We use three CPU platforms in our experiments. Moreover, we are more thorough in finding out the factors behind the performance gap, and we provide tuning suggestions to improve OpenCL performance on CPUs.

Ferrer et al. [30] proposed OmpSs, a programming model based on OpenMP and StarSs [9], which can also incorporate the use of OpenCL or CUDA kernels. They evaluated OmpSs with four benchmarks on three different types of platforms (Intel Xeon server, Cell/B.E., and NVIDIA GPUs), and compared the results with the execution of the same benchmarks written in OpenCL. On the CPU platform, they showed that the OmpSs performance exceeds the OpenCL performance, because in the SMP (symmetric multi-processing) environment, OmpSs relies on the shared memory and avoids all the data copies, while OpenCL still performs data copies explicitly. Our work has also identified this issue, and we use zero copy techniques to solve it [110].

From another perspective, since OpenCL has the cross-platform advantage of programming heterogeneous computing systems, there is ongoing research on using OpenCL on different types of hardware platforms, which inspired us the ways to tune OpenCL performance on CPUs. For example, Fagerlund [28] ran the same AXPY kernel on GPUs and CPUs with different memory access patterns and problem space partitions (i.e., the work-group sizes). The results show that memory access patterns significantly impact GPU and CPU performance, while the use of an appropriate partitioning has a limited positive impact on the overall performance. We have reached a similar conclusion, validated with more extensive experiments.

Stone et al. [123] discussed OpenCL as a parallel programming standard for heterogeneous computing. They summarized the architecture characteristics of three microprocessor families (multi-core CPUs, GPUs, and the Cell/B.E. processor). They related the architecture characteristics, such as SIMD units and memory hierarchy, to the OpenCL platform characteristics. In this chapter, we have also analyzed CPU hardware features to understand the OpenCL performance.

Overall, our work covers a diverse range of applications and different CPU platforms. We provide more insights into the reasons behind OpenCL's performance behavior, and provide efficient tuning suggestions for improving the performance of OpenCL applications on CPUs.

2.6 Summary

OpenCL is designed as a unified programming model for different hardware processors like GPUs and CPUs. Just because OpenCL is increasingly popular for programming GPUs, its potential for performing well on CPUs should not be overlooked. In this chapter, we present an application-centric evaluation of OpenCL on multi-core CPUs, identifying three categories of factors that significantly impact OpenCL performance: (1) incorrect GPU programming style, (2) parallelism granularity, and (3) OpenCL compilers. We further show how these factors can be tuned to improve performance: (1) to remove GPU-friendly coding, users need to tune the memory access patterns and the use of local memory, (2) to address the parallelism granularity issue, users have to tune the granularity systematically to either enlarge its positive impact or avoid its negative impact, and (3) to deal with immature OpenCL compilers, a friendly interface is needed for easily enabling/disabling certain compiler optimizations and checking the resulting performance. Addressing these performance impact factors has allowed OpenCL to achieve good performance on CPUs. This work, combined with previous work [29] (which demonstrates OpenCL is competitive for GPUs), shows that OpenCL is a suitable programming model for CPU and GPU programming. Thus, we adopt OpenCL as the main programming model for programming heterogeneous platforms in the following chapters.

Chapter 3

Accelerating Imbalanced Applications on Heterogeneous Platforms: A Workload Partitioning Framework

In this chapter, we investigate a solution to accelerate *imbalanced applications* on heterogeneous platforms. We observe that, while most applications are currently using homogeneous platforms, imbalanced applications can benefit from the platform heterogeneity. We present *Glinda*, a workload partitioning framework for accelerating imbalanced applications, and show how it improves performance by efficiently utilizing heterogeneous computing resources.

GPUs (Graphics Processing Units) and GPGPU programming (General-Purpose GPU programming) keep gaining popularity in parallel computing [96]. Multiple applications with massive parallelism (e.g., image processing, games, big graph processing, scientific computation) have been significantly accelerated on GPUs [47, 60, 98, 130]. The leading GPU vendors, NVIDIA and AMD, have been continuously releasing new GPU products and updating development tools, aiming to improve GPU computing and make these applications run even faster.

With the rise of GPGPU programming, heterogeneous computing integrating GPUs and CPUs has also become attractive. In this case, the application is divided to run on different processors in a cooperative way, taking the advantage of both the GPU and the multi-core CPU. In addition, single chip hybrid CPU+GPU architectures, such as Intel Sandybridge [53] and AMD Fusion [4], have been launched, becoming a strong incentive for heterogeneous computing.

From the application perspective, applications that achieve high performance on GPUs are usually massively parallel and *balanced* (i.e., each data point of the parallelization space has a relatively similar computation workload, Figure 3.1 (a)). However, applica-

tions can have *imbalanced* workloads (Figure 3.1 (b)), and such cases are not rare. For example, many simulation based scientific applications use a variable time step as a technique to ensure sufficient simulation accuracy, effectively generating different workloads for different simulation points. Note that this imbalance is typically generated by some sort of data-dependent behavior, making imbalanced applications a subset of irregular applications.

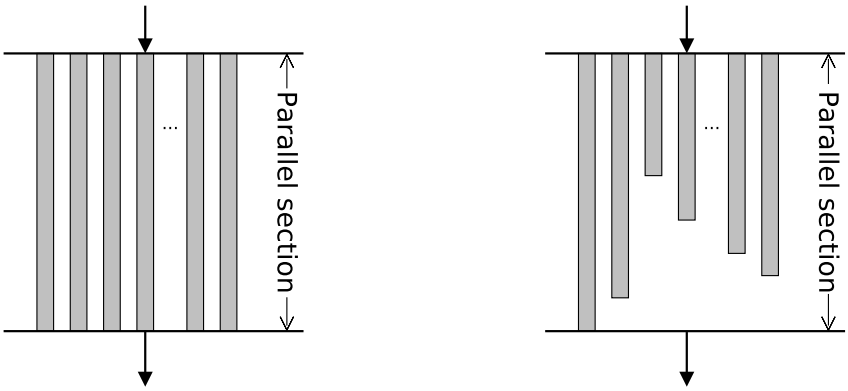


Figure 3.1: Balanced and imbalanced applications. The bar represents the workload (and the life time) of each data point in the parallel section.

In a balanced application, the whole computation can be evenly distributed among all processing cores. They start and finish the computation at a similar pace, ensuring high core occupancy and utilization. Thus, a homogeneous (massively) parallel platform (a GPU or a multi-core CPU) is suitable for this case. When an application has an imbalanced workload, high performance is not always achievable. The utilization of a homogeneous platform will be greatly diminished by the heavier workload part. In this situation, a heterogeneous platform offering a mix of coarse- and fine-grained hardware parallelism and suitable programming tools are desirable [25, 76, 78]. In our work, we use OpenCL as the programming model for heterogeneous platforms. The user has the freedom to partition the workload into different "tasks", and allocate them to different hardware devices.

Still, in order to get the optimal performance, the user needs to tune the workload partitioning according to the workload characteristics and the given heterogeneous platform. As the application workloads can be varied and the heterogeneous platforms can be different, a specific tuning for one execution scenario may not be the best for another. Thus, in this chapter, we focus on the design and implementation of a framework to automatically tune the workload partitioning, aiming to maximize application performance and hardware utilization.

We start from a case study (acoustic ray tracing) of imbalanced applications (Section 3.1). Based on this case study, we define the objectives and requirements of the Glinda framework, and propose its design (Section 3.2). The key functions of Glinda include detecting the application workload characteristics, mapping the right parallel solution on the right hardware configuration, and auto-tuning the optimal workload partitioning (Section 3.3). After implementation, we evaluate Glinda on the same application, and find that it does improve the performance of the application (Section 3.4).

Our main contributions are as follows: (1) we provide a first high performance tunable implementation of acoustic ray tracing for heterogeneous platforms; (2) we further generalize it to three parallel solutions in OpenCL enabling imbalanced applications to utilize CPUs, GPUs, or both; (3) we develop an automated workload partitioning and distribution approach, leading to efficient utilization of heterogeneous platforms; (4) we design and implement Glinda, an adaptive framework which is able to tackle different imbalanced applications, and select the most suitable parallel solution and hardware configuration for a given imbalanced workload.

3.1 Analyzing Imbalanced Applications: A Case Study

In this section, we present an example of an imbalanced application: acoustic ray tracing. We present a detailed analysis of this case study, from its physics to its parallelization, implementation, and achieved performance.

3.1.1 Acoustic Ray Tracing: the Physics

Acoustic ray tracing is a computational method that models the propagation of an acoustic wave front. Acoustic rays are known to refract, i.e., they change direction and the ray path becomes curved due to varying sound speeds. This particular ray tracing application is developed to calculate the sound propagated from an aircraft to a receiver, and consecutively auralize the sound at the receiver position [6]. At NLR (Dutch National Aerospace Laboratory), results from such a calculation are presented in VCNS (Virtual Community Noise Simulator) [104] which allows an assessment of aircraft flyover noise with different atmospheric conditions in a virtual environment.

Acoustic ray tracing applies Snell's law of refraction [107] to calculate the change in ray direction as the medium properties vary. As the sound ray moves through different media, the incident ray (I) is refracted (changes direction) and becomes the transmitted ray (T). A reflected ray (R) is taken into account if the medium boundary is a ground surface or if total internal reflection occurs. The vector N is the unit normal in the upward

direction. After multiple algebraic manipulation, the transmitted ray is established as

$$\mathbf{T} = \eta_{it}\mathbf{I} + \left(\eta_{it} cos\left(\theta_i\right) - \sqrt{1 + \eta_{it}^2\left(cos\left(\theta_i\right)^2 - 1\right)}\right)\mathbf{N} \tag{3.1}$$

$$\eta_{it} = \frac{sin\left(\theta_t\right)}{sin\left(\theta_i\right)} = \frac{v_t}{v_i}$$

where η_{it} is the index of refraction, θ_i and θ_t are the incident and transmitted angles, respectively, and v_t/v_i is the ratio of the acoustic propagating velocities in the different media.

The entire ray propagation path is determined by a time-based simulation, where Equation 3.1 is applied for every time step. This procedure is repeated for multiple rays emitted by a point source at different initial launching angles.

Due to a prevailing headwind at a particular altitude, rays are successively curved upwards and downwards if the source is at the same altitude. Such a situation is referred to as an acoustic duct. Figure 3.2 illustrates ducted rays which occur due to an indentation (headwind component) in the sound speed profile. This indentation is approximated as a sine shaped wind. The strength of the headwind determines the depth of the duct, the stronger the wind the more rays are captured in the duct. If a source is out of the duct region, i.e., above 600 m or lower than 400 m, rays are not trapped in the duct. In VCNS, an aircraft can fly an arbitrary trajectory. If the aircraft is encountering a duct along its trajectory, these ducted rays will occur.

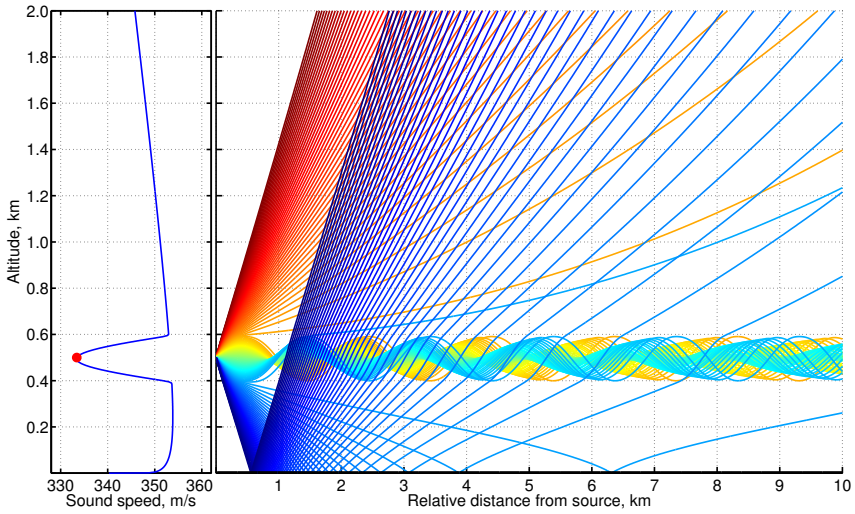


Figure 3.2: Acoustic ray propagation. Left: the sound speed profile with a headwind from 400 m to 600 m altitude, resulting in a duct. Right: the ray propagation profile for all the simulated rays from the source (red dot) at 500 m altitude.

Several atmospheric scenarios are simulated to study the impact of ducted rays. The duct depth, i.e., the max strength of the headwind, is increased from 0 m/s to 10, 20 and 40 m/s while the source remains in the duct. Next, the source is also placed at an altitude of 1000 m out of a duct to simulate the effect of increasing altitude of the aircraft.

The local time step along the ray path controls the distance travelled by the ray, and it is an important factor for both the accuracy and the speed of the simulation. The algorithm uses a variable time step according to the ray location and orientation. The time step is reduced when a ray is near the ground or travels at a shallow angle and is susceptible to refractive effects. This refinement is necessary to increase the resolution and fidelity in the VCNS auralization. The general (coarse) time step is 0.01 s and can be reduced until it reaches the (fine) time step of 0.0005 s.

In VCNS, in order to mimic real-time scenarios, the ray tracing results must be available at every 20 ms for the "in the loop" audio processing. If this criterion cannot be met, VCNS uses an intermediate solution by feeding ray tracing updates when available, and using extrapolation otherwise. Such an approach would relax the calculation time criterion, making a latency of 100 ms acceptable (referred as "soft" real-time requirement).

3.1.2 Acoustic Ray Tracing: the Application

Acoustic ray tracing uses two nested loops to manage the simulation. The outer loop traverses all the rays launched at different angles. The inner loop calculates the ray path propagation (Equation 3.1) over the time steps, and controls the time step resolution. In each time step, the application has to save the intermediate values for all the simulated rays.

As the application chooses a variable time step, the time step of a ray is adjusted dynamically when the ray meets certain special conditions. In the sequential simulation, this solution shortens the total simulation time, while still ensuring sufficient accuracy. However, it also makes the simulated rays have varied lifetimes, as rays that pick finer time steps "live longer" (i.e., their propagation will be interesting for the application for a longer time). Such uneven ray propagation times result in an imbalanced workload distribution over the rays. Figure 3.3 illustrates an acoustic ray tracing workload distribution. A simulated ray is identified by its ray id starting from 0 to ID_{max}, which corresponds to its launching angle ranging from -90 degree and 90 degree with respect to the horizontal. The workload of a ray is measured by the number of simulation iterations. In the shown case, most rays finish their simulation quickly (within 4K iterations), except the rays that meet the time step refinement conditions. These rays run more iterations (up to 60K iterations), and form a narrow "peak" out of the "bottom" in the workload distribution.

In the application, the user sets the atmospheric conditions, the number of simulated rays, and the time step accuracy requirement as input, and determines which parts of the

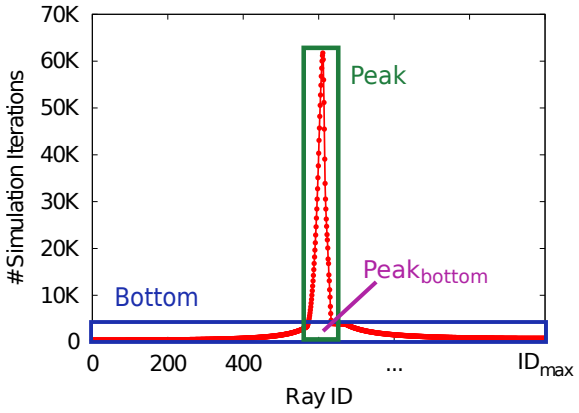


Figure 3.3: An imbalanced ray tracing workload. The workload of a ray is measured by #simulation iterations. Most rays finish simulation quickly (the bottom part). Several rays meet the time step refinement conditions and run much more iterations (the peak part).

data needs to be output for auralization. The original MATLAB code can only simulate around one hundred rays, and needs several minutes to process them. We translated it into sequential C++ code, and parallelized it using OpenCL on a platform with one GPU and one multi-core CPU.

Parallelization

Because ray paths do not depend on each other in ray propagation, all ray paths can be calculated in parallel. This trivially parallel computation is particularly suitable for GPU or multi-core CPU parallelization. Also, taking its imbalanced workload feature into consideration, we design two different parallelization approaches for the application: a homogeneous approach and a heterogeneous approach.

The homogeneous approach exploits data level parallelism. It is designed for using only the CPU or the GPU. As the workload distribution is imbalanced, it is very likely that, soon after starting, the execution will get into an undesirable situation: a few processing cores still have a lot of work to do, while the rest (a majority) are idle. This seriously limits the hardware utilization, leading to low concurrency on the CPU and low occupancy on the GPU.

The heterogeneous approach adds task level parallelism. According to the workload distribution, the whole computation is divided into a bottom part and a peak part (see Figure 3.3). In the bottom part, most rays have relatively even workloads, and therefore it is beneficial to use the GPU for acceleration. For the peak part, only a small portion of the rays (the ones that have a long life) is still in processing, so the CPU is more suitable for computing this part. By making use of both the GPU and the CPU, the heterogeneous approach is able to improve hardware utilization and application performance.

Implementation Details

Based on the two parallelization approaches, we implement three ray tracing parallel solutions using OpenCL. **Ocl.AllCPU** and **Ocl.AllGPU** use the homogeneous approach, and are optimized to run on the CPU and the GPU, respectively. One of the important differences that appears when targeting different architectures is the memory access pattern that the GPU and CPU versions use: the GPU will create and use the data structures in column-major order to improve memory coalescing, while the CPU will use row-major accesses for preserving CPU cache locality. Due to this difference, a transposition phase (TP) is required when data is communicated between the GPU and the CPU. We have chosen to do this transposition on the GPU, before the data is transferred back to the host. This additional transposition, as well as the data movement from the GPU to its host, can add a significant performance penalty to the Ocl.AllGPU solution. **Ocl.Overlap** applies the heterogeneous approach. In order to process the peak and bottom parts in parallel, Ocl.Overlap utilizes a redundant computation scheme by computing the peak_{bottom} part (see Figure 3.3) on both the GPU and the CPU[1]. The final result of the peak part is written by the CPU after finishing the task level parallel section to ensure application correctness. Of course, the same transposition phase is necessary for the data computed on the GPU to be correctly used further on the CPU.

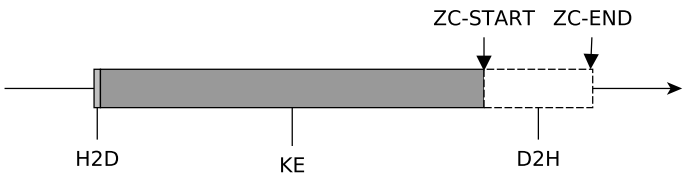
We note that, because we use OpenCL, the same structure is used for all three versions: the host code manages kernel execution and the kernel performs the computation. To target the different configurations (CPU, GPU, and CPU+GPU), we use parameterized code specialization for data transfers and memory access patterns. Thus, when using the GPU, as the host (CPU) and the device (GPU) have separate memory spaces, a real data copy is performed from the host to the device (H2D) and reversed (D2H). For the CPU, as the host (CPU) can directly access the device (the same CPU), zero copy (ZC) is used to avoid unnecessary data copying [110]. Similarly, in the kernel code, the data access pattern is configured to column-major for the GPU version of the kernel, and row-major for the CPU version, thus optimizing the kernel execution (KE).

Our three implementations are presented in Figure 3.4.

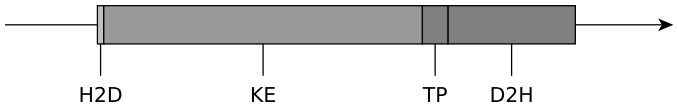
Performance

According to the simulation requirements, several atmospheric scenarios need to be studied (see Section 3.1.1). We show each scenario's workload distribution in Figure 3.5, and identify it by its relative position to the duct and the input wind speed. We see that the wind speed largely affects the workload distribution when the source is placed in the duct. In such scenarios (Figure 3.5(a)-3.5(d)), enhancing the wind speed generates two

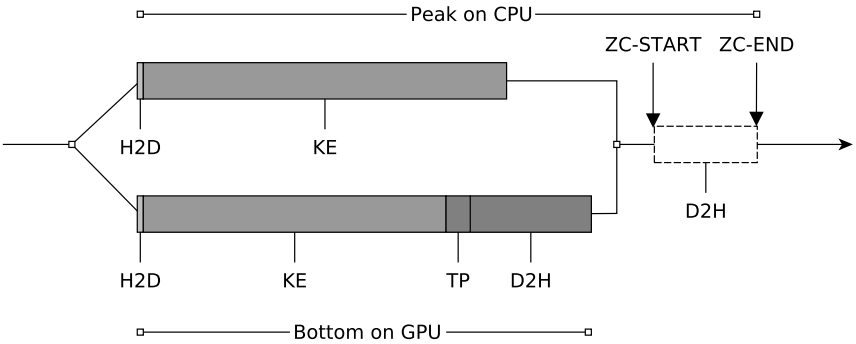
[1]Skipping peak_{bottom} on the GPU has no visible performance improvement in this application. For simplicity, we use a redundant computation scheme.



(a) Ocl.AllCPU



(b) Ocl.AllGPU



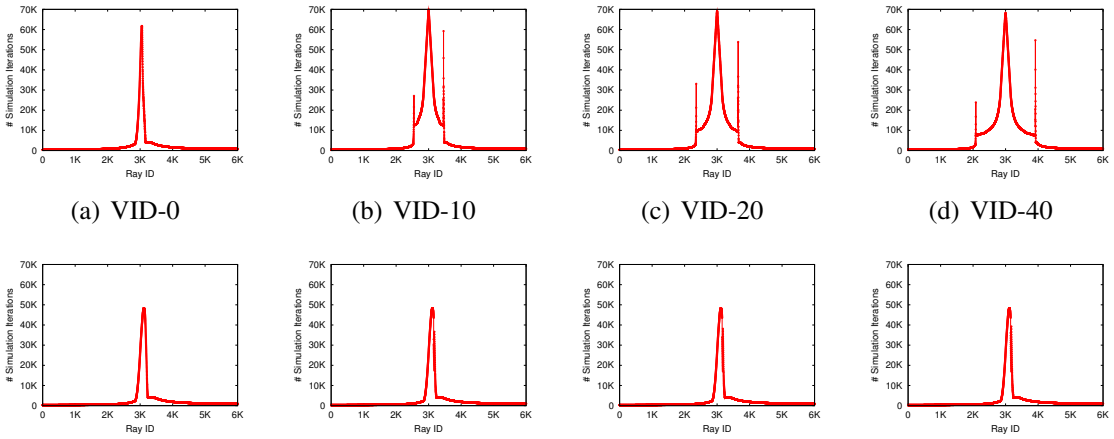
(c) Ocl.Overlap

Figure 3.4: Three ray tracing parallel solutions in OpenCL.

more peaks, and the main peak also becomes wider. When the source is located out of the duct (Figure 3.5(e)-3.5(h)), the workloads with different wind speeds have almost identical distributions. Therefore, we choose VID-0, 10, 20, 40, and VOD-0 as our test cases for the variable time step based ray tracing application.

In order to fully study the performance behavior of different parallel solutions, we also alter the application algorithm by using a fixed *fine* time step and a fixed *coarse* time step. In these situations, the workload distribution becomes flat at the maximum or minimum number of iterations. As the peak disappears, all the test cases have the same workload distribution, therefore we only test the "in the duct" case and the "out of the duct" case with zero headwind speed, i.e., FID-0, FOD-0, CID-0, and COD-0.

To summarize, we have three test categories (*Variable*, *Fine*, and *Coarse*) and nine test cases in total.



(a) VID-0 (b) VID-10 (c) VID-20 (d) VID-40

(e) VOD-0 (f) VOD-10 (g) VOD-20 (h) VOD-40

Figure 3.5: The workload of the variable time step based ray tracing scenarios. The duct spans from 400 m to 600 m altitude. "V" represents the variable time step based approach. "ID" and "OD" indicate that the aircraft source is placed in the duct (500 m altitude) and out of the duct (1000 m altitude), respectively. Finally, the number represents the input headwind speed (m/s).

All experiments have been performed on a heterogeneous platform, including a dual-socket Intel Xeon E5645 six-core CPU and an NVIDIA Tesla C2050 GPU. The detailed hardware platform information is listed in Table 3.1. One of the CPU hardware threads functions as the OpenCL host to manage kernel execution on the OpenCL device(s) and data communication between the two sides. The OpenCL kernel is compiled by the Intel OpenCL SDK 2012 on the CPU and NVIDIA OpenCL SDK 4.1 on the GPU, respectively. The host side compiler is GCC 4.4.3 with -O3 option.

We have run all the test cases using each OpenCL parallel solution. Due to the memory limitation (the amount of data generated during the simulation increases proportionally with both the number of rays and the predefined granularity of the time step), each processor can only hold a certain amount of rays in each test category (see Table 3.1). The *coarse* category has less memory requirements, and therefore can process more rays. In our experiment, we use 900 rays (which fits in all categories), and 6,000 rays (6K, which fits GPU only in *coarse*) as our datasets. The number of rays is controlled by the interval between two adjacent rays. The work-group size is set to 128 rays per group. For Ocl.Overlap, the workload is cut at the 4,000th (4K) iteration and the 10,000th (10K) iteration for 900 rays and 6K rays, respectively.

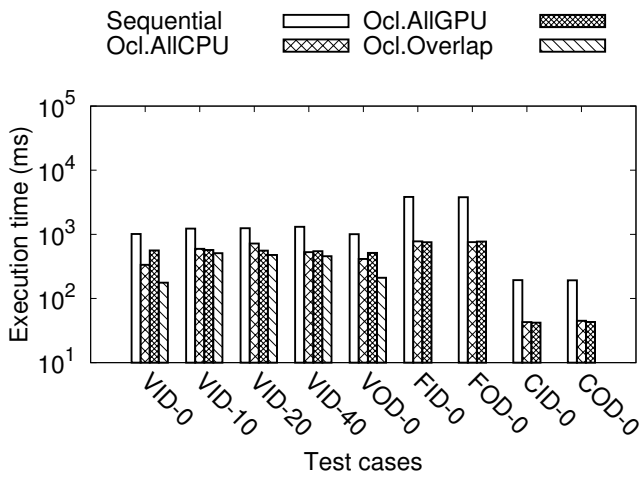
Figure 3.6 shows our experiment results. In both Ocl.AllCPU and Ocl.AllGPU, the performance difference between the *Fine* category and the *Variable* category is rather small, whereas the difference is much larger in the sequential version. This indicates that varying the time step cannot always ensure a better parallel performance, as a uni-

Table 3.1: Details of the computing platform.

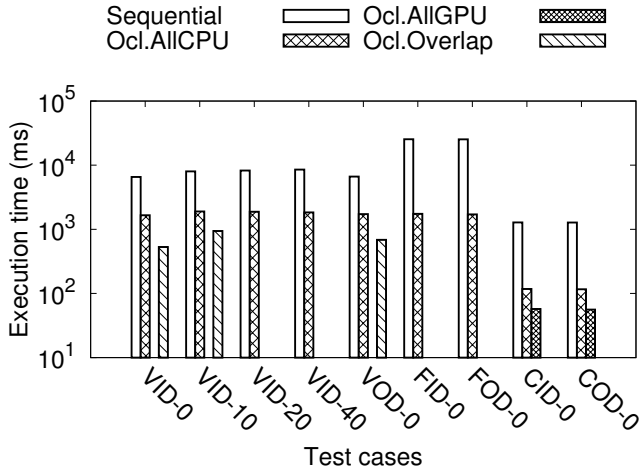
	CPU	GPU
Processor	2×Intel Xeon E5645	NVIDIA Tesla C2050
Core clock	2.4 GHz	1.15 GHz
Core count	24 (hyper-threading)	448
# Processing elements	24	448
# Compute units	24	14
Peak double precision GFLOPS	230.4 Gflops	515.2 Gflops
Memory capacity	24 GB	3 GB
Peak Memory bandwidth	64 GB/s (DDR3-1333)	144 GB/s (GDDR5 1.5 GHz)
Connection with the host	shared memory	PCIe 2.0 ×16
The maximum number of rays due to memory limitation		
Variable	11.2K	1.2K
Fine	11.2K	1.2K
Coarse	66.6K	7.4K

form time step can make the workload evenly distributed over the underlying hardware. Ocl.AllCPU works in all test cases, while Ocl.AllGPU can only run in *Coarse* for the 6K dataset. In addition, Ocl.AllGPU rarely outperforms Ocl.AllCPU, which means that the GPU is not efficiently utilized. As the time step is coarsened (in *Coarse*), the performance is significantly improved for all solutions, but the tracing accuracy is also decreased. Ocl.Overlap is specially designed for the variable time step (VID-0, 10, 20, 40, and VOD-0). It gets the best performance out of the three parallel solutions. Because Ocl.Overlap assumes that only one peak exists in a workload and the cut point is hand picked, it gets wrong tracing results in VID-20 and VID-40 for the 6K dataset (only the first smaller peak is processed on the CPU), and therefore the performance of these two cases is not shown.

In summary, each OpenCL parallel solution has its advantage and disadvantage highly dependent on the workload type and (im)balance. As the user can have very different execution scenarios, accuracy requirements, performance requirements, and hardware platforms, we design and develop Glinda, an integrated framework for choosing the right parallel solution, hardware configuration, and workload partitioning. Furthermore, because the variable time step based simulation is commonly adopted in scientific applications, we want to design Glinda as a general framework that can be potentially used for various imbalanced applications.



(a) Dateset = 900 rays



(b) Dateset = 6K rays

Figure 3.6: The performance of the three OpenCL parallel solutions for (a) 900 rays and (b) 6K rays. The sequential performance on the CPU is given for comparison. Note the log scale on the vertical axis.

3.2 Framework Design

In this section, we generalize the applications we target, define the objectives and requirements of Glinda, and present its structure and components.

3.2.1 Application Generalization

Glinda is proposed for massively parallel applications that have imbalanced workloads. Usually, these applications run on multidimensional datasets, and the parallelization is

along the one dimension that has the most data level parallelism. The kernel of these applications contains one or more loops where the number of iterations is determined at runtime and varies depending on the application algorithm and/or dataset. Figure 3.7 shows the basic structure of the kernel. The time step refinement makes the workload distribution imbalanced. If the time step is decreased to its minimum value in each iteration or data never hits the refinement condition (the time step is kept at its initial value), the workload becomes even at its maximum or minimum.

```
for all(i in data space){
   while(alive[i] && !out of time){
      if(data[i] needs time refinement)
         refine(dt[i]);     //dt is the time step
      update(data[i]);
      t[i] += dt[i];
      if(data[i] has converged);
         alive[i] = 0;
   }
}
```

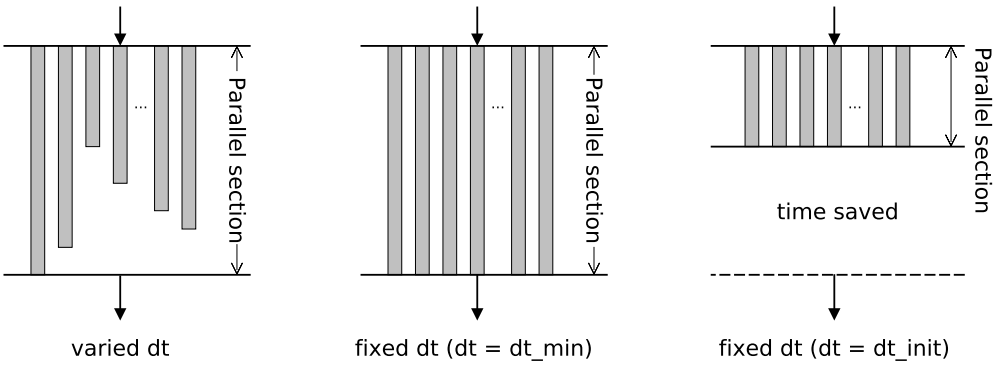


Figure 3.7: The basic structure of the kernel. The variable time step results in an imbalanced workload. The fixed time step results in a balanced workload.

3.2.2 Design Objectives and Requirements

Based on our experience in the acoustic ray tracing parallelization (Section 3.1), Glinda has to meet the following objectives:

1. **High utilization**: Glinda should efficiently utilize all the computing resources on the given hardware platform.

2. **Fast processing**: Glinda should shorten the whole processing time from getting the dataset to generating the output. However, the trade-off between the framework fast processing and accuracy can be controlled by the user.

3. **Applicability**: Glinda is applicable to different imbalanced applications.

4. **Flexibility**: Glinda is able to handle different workload distributions and any input parameters set by the user.

In order to achieve the maximum application performance, Glinda needs to efficiently utilize the underlying hardware. This includes detecting the available hardware resources, choosing the right hardware configuration and parallel solution according to the workload characteristics, and setting the optimal workload partitioning.

In order to shorten the processing time, the workload partitioning should be auto-tuned by heuristic search. The optimal partitioning of each tested scenario is recorded in a repository for future reuse. In addition, using a prediction method to predict the partitioning can prune the search space and speedup the whole processing.

Different applications and different user requirements can generate different workload distributions. When an application test case has an irregular shape in the workload (e.g., Figure 3.5(b)-3.5(d)), additional sorting and de-sorting must be taken to ensure the application correctness.

3.2.3 Framework Overview and Components

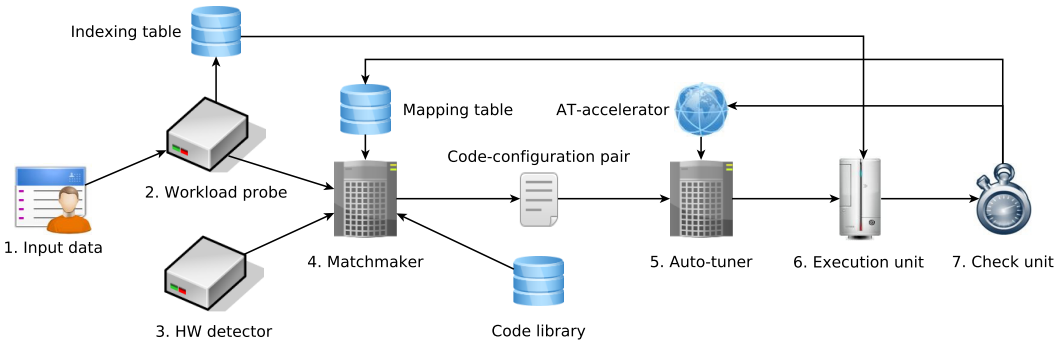


Figure 3.8: The overview of Glinda.

Glinda determines the optimal code version, hardware configuration, and workload partitioning depending on the user-defined input parameters and the available hardware resources. Figure 3.8 shows the overview of Glinda.

1. **The input data** includes the user-defined application parameters and the dataset, which are transferred to the workload probe.

2. **The workload probe** characterizes the application workload distribution by sampling. If multiple peaks exist, sorting the original dataset by the number of iterations is performed and the sorting result is stored in the **indexing table**.

3. **The HW detector** checks the available hardware resources and determines possible hardware configurations. There are three options: (1) running the application only on the CPU (*Only-CPU*), only on the GPU (*Only-GPU*), or using a mix of both (*CPU+GPU*).

4. **The matchmaker** proposes the optimal **code-configuration** pair according to the outputs from the workload probe and the HW detector. It chooses the best code candidate from the **code library** (which has Ocl.allCPU, Ocl.allGPU, and Ocl.Overlap) and the most suitable hardware configuration from the possible options. A code-configuration **mapping table** records previous optimal mapping information that helps the selection.

5. **The auto-tuner** takes the selected code-configuration pair as input, and generates the optimal workload partitioning through auto-tuning. The auto-tuning includes two aspects: (1) in the *granularity* aspect, the auto-tuner finds the optimal parallelism granularity for each device (i.e., the work-group size in OpenCL.); (2) in the *task* aspect, the auto-tuner partitions the workload into a GPU task and a CPU task, and detects the optimal partitioning. In order to further improve performance, a data redistribution on the same device with a different granularity might be needed. An **AT-accelerator** selects the search space and prunes it to speedup the auto-tuning process.

6. **The execution unit** performs the real computation, running the tuned code with the selected hardware configuration. If the workload has been sorted, the output data is also de-sorted according to the indexing table to get the correct output.

7. **The check unit** validates the correctness of the code and records its execution performance. If the code gets correct output and the execution time meets the real-time requirement, the check unit writes the code-configuration mapping pair and the workload partitioning into the mapping table and the AT-accelerator, respectively, for future reuse.

In the next section, we further elaborate on the key components of the framework.

3.3 Key Components of the Framework

The key components of Glinda include probing the workload characteristics, matchmaking the code-configuration pair, and auto-tuning the optimal workload partitioning.

3.3.1 Workload Probing

The workload probe profiles the workload of a user-defined test case of a given application. It consists of a sampling phase and an estimation phase.

The sampling is performed in the data parallelization dimension, the time step dimension, or the combination of the two. In the data parallelization dimension, the workload probe samples the original dataset at a pre-determined sampling rate R_{sp}. In the time step dimension, the initial time step is enlarged by a coarsening coefficient C_{sp}. Higher R_{sp} and C_{sp} make sampling faster, while they also affect the sampling result. Therefore, choosing the right values becomes a trade-off between speed and accuracy, and can be decided by the user.

Algorithm 1 Estimate the workload characteristic

Input: N, t_{thrsh}, dt_{init}, dt_{min}, R_{sp}, C_{sp}, D_i^{sp}
Output: CH = {0, 1, n}

$N_{sp} \leftarrow N/R_{sp}$
$D_{max}^{sp} \leftarrow t_{thrsh}/(dt_{min} \times C_{sp})$
$D_{min}^{sp} \leftarrow t_{thrsh}/(dt_{init} \times C_{sp})$
$D_{est}^{sp} \leftarrow F(D_{max}^{sp}, D_{min}^{sp})$ ▷*get the peak estimation height*

for $i = 1 \rightarrow N_{sp}$ **do**
    if $D_i^{sp} > D_{est}^{sp}$ **then**
        $Index[Counter] \leftarrow i$
        $Counter \leftarrow Counter + 1$
    end if
end for
$P \leftarrow Counter/N_{sp}$ ▷*get the proportion of samples exceeding* D_{est}^{sp}

if $P \geq P_{min}$ AND $P \leq P_{max}$ **then**
    $CH \leftarrow 1$ ▷*peak(s) exist when* $P \in [P_{min}, P_{max}]$
    if $Index$ value is NOT consecutive **then**
        $CH \leftarrow n$ ▷*multiple peaks when* $Index$ *is not consecutive*
    end if
else
    $CH \leftarrow 0$ ▷*regard other cases as flat workloads*
end if

After getting the sampling result (the number of iterations per sample), the workload probe estimates the workload characteristic CH. Depending on the the number of peaks (samples in the peak have more iterations), a workload can be flat, can have one peak, or multiple peaks. An estimated peak height D_{est}^{sp} is derived from the maximum number of

iterations D^{sp}_{max} and the minimum number of iterations D^{sp}_{min} in sampling. The workload probe scans each sample's number of iterations D^{sp}_{i}, and calculates the proportion P of samples whose number of iterations exceeds D^{sp}_{est}. Based on the value of P, the workload probe determines whether a workload is flat or not. If P is quite small or quite large, the workload is still approximately classified as a flat one. By further checking the continuity of the sample indexes, the workload probe determines whether a workload has one or multiple peaks. Therefore, the estimation accuracy depends on how we set D^{sp}_{est}, P_{max} (the upper bound) and P_{min} (the lower bound) of P. The estimation procedure is listed in Algorithm 1, where the dataset size (the total number of work-items), the sample size, the application time bound, the initial time step, and the minimum time step are noted N, N_{sp}, t_{thrsh}, dt_{init}, and dt_{min}, respectively.

Finally, in order to make Ocl.Overlap work correctly, when multiple peaks are detected, the workload probe will sort the original dataset to re-shape the workload into a single-peak form. Sorting will not be applied for the single-peak workload.

3.3.2 Matchmaking

The matchmaker matches the right code version with the right hardware configuration. The matchmaking policy includes two criteria.

- Workload characteristic CH: 0, the workload is flat (balanced application); 1, the workload has a peak (imbalanced application).
- Hardware resource type RT: 0, the hardware platform is a CPU platform; 1, the hardware platform is a CPU+GPU heterogeneous platform (the GPU cannot work in a standalone mode, so the GPU together with the host CPU form a heterogeneous platform).

Table 3.2 summarizes the matchmaking policy.

Table 3.2: Matchmaking policy.

CH	RT	Code	Configuration
0	0	Ocl.AllCPU	Only-CPU
0	1	Ocl.AllGPU	Only-GPU
1	0	Ocl.AllCPU	Only-CPU
1	1	Ocl.Overlap	CPU+GPU

When the workload is flat ($CH = 0$), the matchmaker chooses Ocl.allCPU if the hardware platform is a CPU platform, and Ocl.allGPU if the hardware platform has a GPU. This means that Glinda matches balanced applications with the homogeneous part

of the platform. When the workload has a peak ($CH = 1$), the matchmaker chooses Ocl.Overlap for the heterogeneous platform, and falls back to Ocl.allCPU if the platform only has a CPU. Therefore, for imbalanced applications, Glinda uses all the hardware devices on the platform.

3.3.3 Auto-tuning

After the code-configuration pair is selected, the auto-tuner detects the optimal workload partitioning with the help of the AT-accelerator.

In the granularity aspect, the auto-tuner focuses on each device to get its optimal parallelism granularity. The parallelism granularity is measured by the work-group size WG, i.e., the number of work-items per work-group, in the code. Increasing the work-group size reduces the number of work-groups. A larger number of work-groups provides more flexibility in scheduling, while the switching overhead is also increased. Therefore, the choice of the work-group size is a trade-off. The auto-tuner varies WG (e.g., $WG = 2^n$, n = 0, 1, 2, ..., 10) on each device to detect the optimal one.

Algorithm 2 Auto-tune the partitioning

Input: t_{thrsh}, dt_{init}, dt_{min}, T^{GPU}_{bottom}, T^{CPU}_{peak}
Output: D_{cut}, WI^{start}_{peak}, WI^{stop}_{peak}

$T_{overlap} \leftarrow Max\{T^{GPU}_{bottom}, T^{CPU}_{peak}\}$
$D_{max} \leftarrow t_{thrsh}/dt_{min}$, $D_{min} \leftarrow t_{thrsh}/dt_{init}$
$D_{high} \leftarrow D_{max}$, $D_{low} \leftarrow D_{min}$
$D_{cut} \leftarrow (D_{high} + D_{low})/2$ ▷*search start point*

while $|T^{GPU}_{bottom} - T^{CPU}_{peak}| > \delta$ **do** ▷δ = *1 ms*
    if $T^{GPU}_{bottom} < T^{CPU}_{peak}$ **then**
        $D_{low} \leftarrow D_{cut}$ ▷*Increasing* D_{cut} *is beneficial*
        $D_{cut} \leftarrow (D_{high} + D_{low})/2$
    end if
    if $T^{GPU}_{bottom} > T^{CPU}_{peak}$ **then**
        $D_{high} \leftarrow D_{cut}$ ▷*Decreasing* D_{cut} *is beneficial*
        $D_{cut} \leftarrow (D_{high} + D_{low})/2$
    end if
end while

Get D_{cut}, WI^{start}_{peak}, WI^{stop}_{peak}

In the task size aspect, the auto-tuner finds the partitioning (the "cut point") to divide the workload into a bottom (flat) part and a peak part. Specifically, this means that the

cut point between the bottom part of the workload and its peak is automatically chosen to optimize the computation time. The bottom part will be ideally executed on the GPU, and the peak part will be ideally executed on the CPU[2]. The auto-tuner follows a binary search to get the optimal cut point D_{cut}. Because the total execution time $T_{overlap}$ is decided by the larger one of the GPU execution time T^{GPU}_{bottom} and the CPU execution time T^{CPU}_{peak}, the goal is to make the two perfectly overlap, such that the bottom part and the peak part are processed efficiently in parallel. Therefore, the auto-tuner compares T^{GPU}_{bottom} and T^{CPU}_{peak}, increases or decreases D_{cut} based on the comparison result, until the balance is reached. In practice, we use a threshold δ to control the auto-tuning. The AT-accelerator sets the search start point at the half between the maximum number of iterations D_{max} and the minimum number of iterations D_{min}. Algorithm 2 explains the auto-tuning process. Finally, the boundary of the peak (corresponding to two work-items WI^{start}_{peak} and WI^{stop}_{peak}) is obtained according to the detected D_{cut}.

For some applications, the peak boundary can be predicted using the physics principle, which can be implemented in the AT-accelerator and supersede the auto-tuner.

In addition, it is possible that during the execution on a single device, some work-items are still "alive" while the others have already "died". It can happen on Ocl.AllGPU and Ocl.AllCPU with a peak workload, or on Ocl.Overlap when the bottom or peak still has a gentle slope. This is a case for workload redistribution. The auto-tuner chooses a smaller WG when the number of alive work-items is below the redistribution width RW. RW is predicted at the half of the total allocated work-items on that device. Each time, WG is reduced by 2 to compare if the execution time with redistribution is smaller than that without redistribution. If this holds, the redistribution scheme with a new detected WG is adopted.

3.4 Experimental Evaluation

In this section, we evaluate Glinda in terms of application performance improvement and framework overhead. We use the acoustic ray tracing application (Section 3.1) with the same test cases and experimental setup.

In workload probing, the sampling correctly preserves a workload shape when R_{sp} and C_{sp} are set smaller than 20 and 5, respectively. Decreasing R_{sp} and C_{sp}, the sampling speed is decreased by the same factor, while the accuracy is improved. A practical estimation of the workload characteristic is ensured by setting D^{sp}_{est} at the half between D^{sp}_{min} and D^{sp}_{max}, and setting P_{min} and P_{max} at 5% and 50%, respectively. The maximum workload probing speed (from sampling to getting a correct CH) is 0.003 ms per ray. As the detection results show that VID-10, 20, 40 have multiple peaks, their datasets are

[2]The task size auto-tuning is used for the Ocl.Overlap version.

further sorted, requiring an extra 4.6 ms for 6K rays and only 0.6 ms for 900 rays.

In the matchmaking phase, each test case gets a parallel code version and a hardware configuration according to the matchmaking policy. We list the mapping results in Table 3.3, and we use the selected code-configuration pairs for auto-tuning and execution.

Table 3.3: Matchmaking results.

Datasets	Test cases	CH	RT	Code	Configuration
900 rays	VID-0, 10, 20, 40	1	1	Ocl.Overlap	CPU+GPU
	FID-0, FOD-0	0	1	Ocl.AllGPU	Only-GPU
	CID-0, COD-0	0	1	Ocl.AllGPU	Only-GPU
6K rays	VID-0, 10, 20, 40	1	1	Ocl.Overlap	CPU+GPU
	FID-0, FOD-0	0	0*	Ocl.AllCPU	Only-CPU
	CID-0, COD-0	0	1	Ocl.AllGPU	Only-GPU

* The dataset cannot fit in the GPU memory, so only the CPU can be used.

In the auto-tuning phase, Glinda finds the best parallelism granularity for the GPU (WG = 16 - 64) and the CPU (WG = 1) for each test case. We see that they are different from the original hand picked ones. To determine the cut point for task decomposition, we have two approaches: (1) directly using auto-tuning, and (2) using physics-based prediction instead of auto-tuning. Both of them may have important effects on both the application performance and the framework overhead. The physics-based prediction is able to get all peaks in a workload using the duct principle, and therefore sorting is not needed in this approach. We compare the auto-tuning and the physics-based prediction in Table 3.4. The ray tracing performance difference between the two enlarges with the increase of the wind speed (In VID-0 and VOD-0, the two have similar performance). This is because the auto-tuning finds a higher cut point based on the sorted dataset, where the bottom on the GPU and the peak on the CPU have closer execution times. The physics-based prediction determines a lower cut point which ensures it gets the correct peaks, but the peak execution time on the CPU is much larger. In terms of overhead, the auto-tuning process needs 4 to 7 rounds of searching (depending on the workload), whereas the physics-based prediction does not. Glinda is flexible to utilize either approach according to the user requirement.

We have also tested the workload redistribution, but we achieved no significant improvement. Therefore, we omit the results for redistribution tuning.

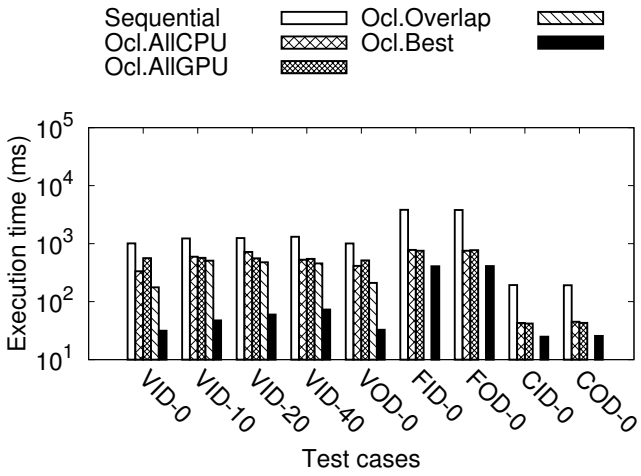
Figure 3.9 shows the tuned performance (the best performance) of each test case for the selected code-configuration pair (Table 3.3). Comparing to the sequential version and the original parallel versions (without using Glinda), Glinda significantly improves ray tracing performance. The average performance speedup against the original parallel performance is 11×, 12×, 7× for Ocl.AllCPU, Ocl.AllGPU and Ocl.Overlap, respectively.

Table 3.4: The comparison between auto-tuning and physics-based prediction. The peak width $PW = WI_{peak}^{stop} - WI_{peak}^{start}$. The ray tracing execution time (ms) is listed.

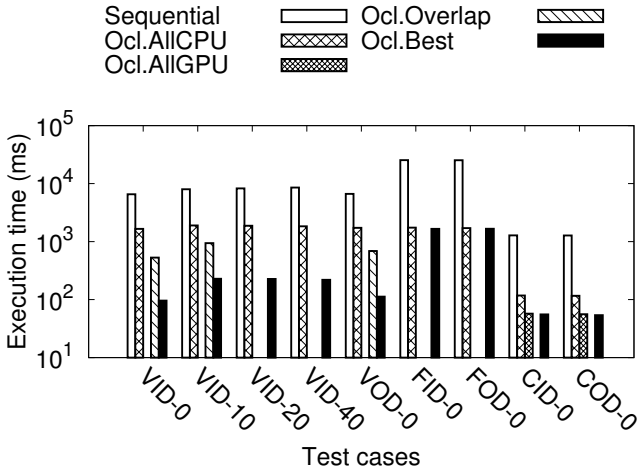
	Auto-tuning			Physics-based prediction		
900 rays	PW	D_{cut}	Execution Time	PW	D_{cut}	Execution Time
VID-0	106	2,870	31.5	102	2,964	31.8
VID-10	143	4,375	47.4	230	1,853	64.3
VID-20	197	6,562	60.0	287	1,641	88.0
VID-40	277	7,109	73.0	372	1,403	91.2
VOD-0	102	3,281	32.7	122	2,715	36.7
6K rays	PW	D_{cut}	Execution Time	PW	D_{cut}	Execution Time
VID-0	271	6,562	96.2	335	3,975	97.4
VID-10	709	13,944	228.4	1,066	3,529	289.5
VID-20	699	13,944	226.9	1,444	3,076	357.4
VID-40	659	13,944	219.7	1,992	2,626	473.9
VOD-0	304	8,203	112.7	434	3,926	122.5

There are three sources of overhead in Glinda: the sampling, the sorting, and the auto-tuning. However, depending on the user scenarios, the overhead will vary. For example, decreasing the sampling rate will increase the overhead, but will also give a more accurate sampling result. Similarly, auto-tuning can be time-consuming, but if it can be replaced by the physics-based prediction, this might reduce both the sorting and the auto-tuning overhead to zero. If, however, the physics-based prediction is not possible, using the auto-tuning will add some overhead, but will also maximize the performance. We also note that we expect VCNS to require running the same test scenarios repetitively (i.e., the same acoustic conditions), in which Glinda does not need to repeat the whole process of sampling, sorting, and auto-tuning—but rather fetch the necessary configuration from the repository. Therefore, for many of these repetitive experiments, Glinda mostly exhibits a one-time overhead, for its first execution for a new acoustic test scenario.

Before applying Glinda, only the coarse time step meets the soft real-time requirement (100 ms) for both datasets. After using Glinda, for the smaller dataset (900 rays), the variable time step meets the soft real-time requirement with higher accuracy, and the coarse time step meets the real-time requirement (20 ms). In VCNS, these 900 rays are sufficient for mimicking real-time scenarios. For the larger dataset (6K rays), the variable time step meets the soft real-time requirement in two (VID-0 and VOD-0) out of five test cases, which means if such a large number of rays are needed, more compute devices should be added in the platform to further accelerate the application.



(a) Dateset = 900 rays



(b) Dateset = 6K rays

Figure 3.9: The best performance (Ocl.Best) of each test case after using Glinda. The original performance is given for comparison. Note the log scale on the vertical axis.

3.5 Related Work

Multiple studies in recent years have proved that integrating both GPUs and CPUs on heterogeneous platforms leads to improved application performance, hardware utilization, and power efficiency [33, 41, 44, 128].

Parallel solutions for utilizing hybrid computing resources usually focus on workload partitioning and/or workload scheduling. Grewe et al. [37] proposed a static task partitioning approach based on code features. They built a predictive model that maps code features to partitions through machine learning. Their approach is relatively fast, but unlike ours, it is not flexible to changes in runtime like input parameters and datasets.

Merge [76] and the work in [125] adopted a manual approach where the right task partitioning is determined or tuned by the programmer. In both cases, the applicability of the system is constrained, and the burden on the programmer is relatively high. By comparison, our approach applies auto-tuning. Qilin [78] relates to our work because it also uses an automatic approach to find the best workload decomposition. While they analytically derive the decomposition from historical performance data, we take the workload characteristics into consideration and use a more accurate systemic tuning approach.

Research in [8,25,59,97] adopted dynamic task scheduling schemes to distribute tasks to different compute devices. The main idea is to transparently take advantage of heterogeneous platforms without the support of the programmer or the auto-tuner. Because the hardware details may not be fully taken into consideration, a wrong scheduling policy can lead to poor performance as shown in these papers. In our work, we use workload partitioning to ensure maximized performance.

All the above mentioned studies (except [37, 76]) use separate GPU and multi-core CPU programming models (e.g., CUDA on GPUs and OpenMP on CPUs). Therefore, they cannot work on the integrated CPU+GPU shared memory hardware architectures (e.g., Intel Sandybridge, AMD Fusion). In our framework Glinda, we use OpenCL as a unified parallel programming paradigm, enlarging the portability of Glinda over various heterogeneous platforms without significant performance loss.

3.6 Summary

Applications with imbalanced workloads fit naturally on heterogeneous platforms, where tasks can be partitioned according to hardware architecture characteristics, and assigned to run on different hardware devices. In this chapter, we propose Glinda, a workload partitioning framework for accelerating imbalanced applications. To make the whole procedure systematic and automatic, we have developed a series of techniques as components in Glinda, including workload characteristics probing, hardware detection, code-configuration matchmaking, and partitioning auto-tuning. Based on the first three components, Glinda is able to select the best parallel solution and hardware configuration for a given application and hardware platform. In its fourth component, Glinda adopts an automated workload decomposition approach, where it decomposes the workload into CPU and GPU tasks, and tunes the optimal parallelism granularity and task size for each device. In summary, Glinda significantly accelerates imbalanced applications, improves hardware utilization on heterogeneous platforms, and is flexible enough to deal with different user requirements, application scenarios, and hardware environments.

We also notice that the most time-consuming phase in Glinda is the auto-tuning. Therefore, in Chapter 4, we aim to develop a generic prediction method, rather than the physics-based prediction used in this chapter, to speedup the workload partitioning.

Chapter 4

Optimizing Workload Partitioning: A Prediction Method

In Chapter 3, we have shown how heterogeneous platforms can be efficiently exploited by imbalanced applications. With clever auto-tuning, we have shown how to partition the imbalanced workload to fit the usage patterns of the processors. In this chapter, we present a flexible and adaptive method that predicts the optimal partitioning. By replacing auto-tuning with prediction, we optimize the partitioning process.

Massively parallel applications are typical workloads for GPUs, under the assumption that the parallelism of the application and the hardware platform match. However, not all massively parallel applications are as regular as needed to fit this profile. For example, imbalanced applications (i.e., applications with an imbalanced workload) turn out to be more of a challenge than a fit for GPUs.

Such applications can be found in domains of scientific simulation, numerical methods, and graph processing [6, 34, 83, 100, 129]. Imagine a scientific simulation using a variable time step as a technique to ensure sufficient accuracy: this approach generates different workloads for different simulation points. Similarly, many real graphs have power-law degree distributions, where higher-degree nodes require more processing in algorithms like breadth first search (BFS) [47] or finding connected components [99]. These relatively few data points that require more computation drastically limit the GPU utilization and eventually lead to poor performance.

To accelerate imbalanced applications, we have proposed the exploitation of heterogeneous platforms, i.e., we combine the individual strength of the GPU *and* the CPU (the host processor). By enabling workload partitioning, we might obtain improved performance. However, the best performance can only be achieved when the workload is partitioned to best utilize each processor.

The optimal workload partitioning is not trivial: hardware capabilities, workload char-

acteristics, and the amount of data that needs transferring must all be taken into account. The auto-tuning method presented in Chapter 3 is an option, but it can be time-consuming (and it keeps the target heterogeneous platform occupied to run the search). Understanding the application and hardware, either analytically (depending on what the application does) [78] or through statistical modeling [37, 67] can also lead to efficient partitioning, but these approaches are too complex, often time-consuming, or less suited for imbalanced applications.

In this work, we propose *a model-based prediction method for imbalanced workload partitioning*. Our method detects whether partitioning is necessary and, if so, predicts the suitable partitioning point(s) for the heterogeneous platform. In this method, we build a *partitioning model*, based on a given fitting criterion (execution time in this work), that matches a quantitative model of the application (*a workload model*) with the *hardware capabilities* of the given platform. By using the workload model, we quantify the application workload and its characteristics, as well as the data transfer between processors. On the platform side, we estimate hardware capabilities by using a low-cost profiling. This profiling is necessary to cope with the sensitivity of a platform to the application, to the problem size, and to the drivers and compilers. By solving the partitioning model for the given fitting criterion, our method predicts the optimal workload partitioning.

To evaluate our partitioning method, we use synthetic benchmarks (1295 imbalanced workloads), and a real-world application (10 imbalanced workloads), all implemented in OpenCL. We run experiments to evaluate the partitioning impact on performance, and validate the accuracy and adaptiveness of our method. Our results show that partitioning improves application performance by up to 85% compared to the use of a single processor (a CPU or a GPU). The optimal partitioning is correctly predicted in more than 90% of the cases, on various platforms and workloads.

To examine the usability of the method, we applied it to a real-world case study [6], showing a systematic recipe that leads to efficient partitioning. This step-by-step procedure can be repeated for other imbalanced applications.

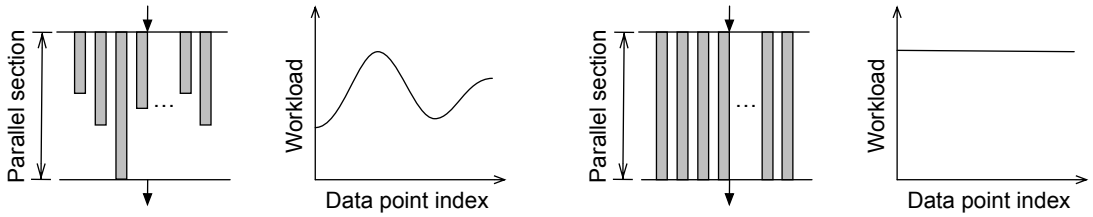
The main contributions are as follows: (1) we develop a quantitative workload modeling for imbalanced applications; (2) we design a partitioning model that fits the workload model with the platform's hardware capabilities; (3) we design and implement a prediction method that, by solving the partitioning model for a user-given criterion, predicts the optimal workload partitioning for a given platform; (4) we present a detailed recipe for applying the method to real-life applications.

This chapter is organized as follows. We analyze imbalanced workloads in Section 4.1. Section 4.2 then describes the model-based workload partitioning method in detail. The experiment evaluation is given in Section 4.3, followed by Section 4.4 with the real-world case study. We discuss related work in Section 4.5. Finally, we conclude the chapter in Section 4.6.

4.1 Imbalanced Workloads

We define an application having an *imbalanced workload* when it has massive parallelism, but varied workload per data point. A data point is an independent element in the application parallelization space. The *workload* of a data point is a measurable quantity, where the choice of metric depends on the application scenario. For example, scientific simulation can use the number of iterations [6], numerical methods can use the number of floating-point operations [83], and graph processing can use the number of neighbor nodes (the degree) [34].

Usually, an application gets an imbalanced workload due to either the algorithm and/or the dataset. For the algorithm case, for example, in a scientific simulation, the user may be interested in some simulation points, thus making these points run more iterations and perform more computations. For the dataset case, for example, in graph traversal, because the graph can be quite asymmetric, the nodes with higher degree need more processing than the others. In either case, there are two common features in imbalanced workloads: (1) in the data-point dimension (the parallelization dimension), there is no dependency between two data points; (2) in the workload dimension, the workload of each data point is varied, determined at runtime[1]. By contrast, in a *balanced workload*, each data point has a similar workload known at compile time or even before that. Figure 4.1 illustrates the difference between balanced and imbalanced workloads.



(a) Imbalanced workload (b) Balanced workload

Figure 4.1: (a) Balanced workload. (b) Imbalanced workload. The bar chart shows the life time of each data point in the parallel section. The line chart shows the distribution of the workload per data point. Note that the distribution is in fact discrete. We show it (also for the figures in the rest of this chapter) as a continuous distribution to simplify the figure presentation.

For imbalanced workloads, achieving high performance on GPUs is difficult, as the imbalanced part hinders the utilization of the platform. In such situation, using both GPU and CPU with each processor taking a suitable partition will lead to better hardware utilization. Therefore, to improve performance and to optimize the partitioning process,

[1]The workload per data point cannot be determined before execution, but the runtime behavior is constant once the algorithm and/or the dataset are fixed.

we propose a prediction-based workload partitioning method that predicts the optimal partitioning for imbalanced workloads.

4.2 Prediction-Based Workload Partitioning

In this section, we present the design and development of our prediction-based workload partitioning method.

4.2.1 The Big Picture

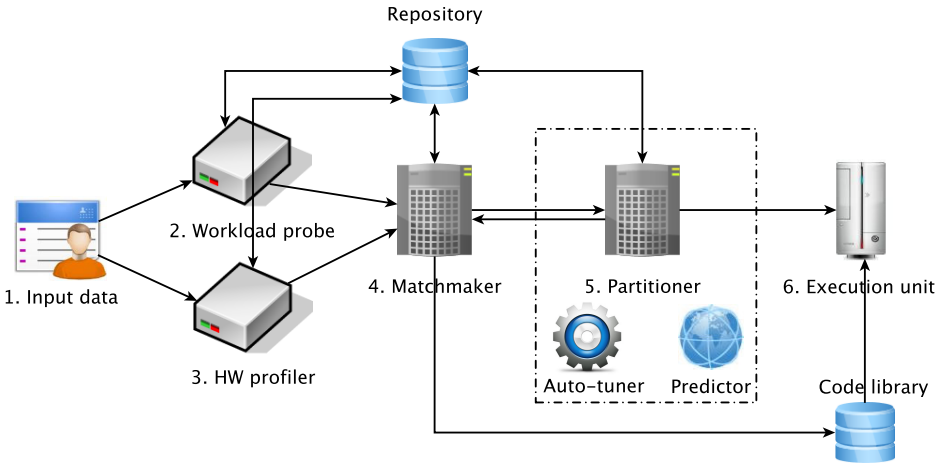


Figure 4.2: The entire partitioning process illustrated in Glinda (Chapter 3).

Figure 4.2 shows the entire partitioning process. (1) The user inputs a parallel application, its parameters (e.g., problem size), and its dataset, which are delivered to the *workload probe* and the *HW profiler*. (2) The *workload probe* characterizes the application by its workload features and generates a *workload model*. (3) The *HW profiler* detects the available processors and profiles the processors to estimate the *hardware capabilities*. (4) The *matchmaker* proposes the hardware configuration (e.g., CPU+GPU), and asks the partitioner to determine the optimal partitioning. (5) The *predictor* (which replaces the auto-tuner) uses the workload model and the hardware capabilities to predict the optimal partitioning. The output generated in (2)–(5) are stored in *repositories* for reuse, further accelerating the partitioning process. (6) Finally, the *execution unit* (the target heterogeneous platform) executes the partitioned workload. The partitioning process can run online, on the same machine as the workload, or offline, on a different machine, which will assist the deployment.

Input applications are parallelized in OpenCL [64], where the host (the CPU) manages data transfers between the host and the devices (the CPU and the GPU). For the CPU device, as the host and the device share the same physical memory, data transfers can be avoided using zero copy [110]. For the GPU device, as the host and the device have separate memory spaces, real data copies are performed, and this data-transfer overhead has to be considered into the workload partitioning.

We note that we choose OpenCL for portability, and we use the same parallel version of the application with architecture-specific code specialization [40, 109, 112], thus limiting the effort the users have to put into porting the application to the CPU (while assuming the GPU code is available but underperforms). To enable partitioning, the host code needs to be modified to support OpenCL multi-device initialization, workload distribution, and output gathering. We implement a set of APIs to simply the coding. We further note that our partitioning method is *not* dependent on OpenCL. If the users want or have different languages for using this method (e.g., using OpenMP for CPU or CUDA for GPU), it is also possible. In our experience, this does not change the methodology, but only shifts the partitioning one direction or another.

4.2.2 The Workload Model

We build a workload model to quantify the workload and its characteristics, which is the starting point of the prediction-based workload partitioning.

Because the workload is imbalanced, we use sorting (i.e., we sort the data points according to their workloads) to reshape the whole workload into a more regular shape. In this way, similar behaving data points will be grouped together, leading to a more intuitive partitioning. Figure 4.3 shows the workload reshaping process. A sorted workload can have one of the three basic shapes: a *plateau* shape, a *peak* shape, or a *stair* shape. We note that more apparently complicated workload shapes can be either approximated to one of the basic shapes or decomposed into the several basic shapes.

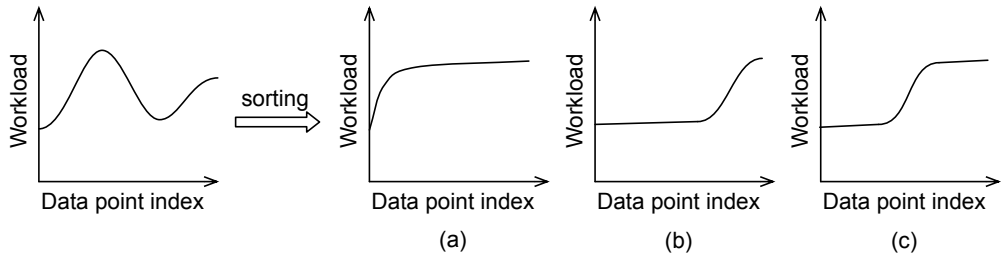


Figure 4.3: The workload reshaping process. An imbalanced workload is sorted into a more regular shape: either (a) plateau shape, (b) peak shape, or (c) stair shape.

From the three basic shapes, an imbalanced workload is best represented by the peak shape. This is because (1) the performance of the plateau-shaped workload is dominated by the plateau part (the flat part), so the imbalanced part will not have much influence on the overall performance; (2) the peak-shaped workload fits the heterogeneous platforms, as the GPU with bulk parallel processing is efficient at processing the flat part, and the CPU with better single thread performance is suitable for processing the peak part (the part with more irregularity); (3) the stair-shaped workload can be approximated by either a shape with a wider peak (thus covered by the peak shape) or a shape with two dominating plateaus (thus becoming a two-phase balanced workload). For now, we approximate this with a wider peak.

Next, we build a workload model for the peak-shaped workload, shown in Figure 4.4. We assume that the workload shape is composed of a horizontal line (representing the flat part) and a slanted line (representing the peak part) with an inflection point ip in between. To quantify the characteristics of the imbalanced workload, we use the following parameters: the problem size (n), the workload in the flat part (base height, noted as bh), the number of data points in the peak part (peak width, noted as pw), and the maximum workload in the peak part (peak height, noted as ph). We also derive two peak characteristics from the workload model: s, the slope of the peak ($s = (ph - bh)/pw$), and α, the ratio of pw to n.

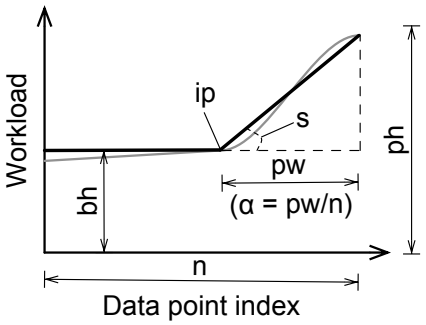


Figure 4.4: The workload model overview for imbalanced workloads.

Thus, the workload of each data point can be presented by a two-phase linear function:

$$w_i = \begin{cases} bh & i = 0, ..., n - pw - 1 \\ s \times (i - (n - pw - 1)) + bh & i = n - pw, ..., n - 1 \end{cases}$$

where w_i is the workload of a data point i, and i is the data point index.

In summary, we form a workload model which separates the flat part and the peak part of the workload with an inflection point ip. Its characteristics parameters are n, bh, pw, ph, s, and α.

4.2.3 The Partitioning Model

The best performance on the platform is achieved when there is a perfect execution overlap between the two processors: $T_G + T_D = T_C$. T_G and T_C are the kernel execution time on the GPU and the CPU, respectively. T_D is the data-transfer overhead added on the GPU[2]. This equation is the criterion that defines the optimal partitioning.

We adopt a two-step quantitative approach to model the execution of the workload. In the first step, T_D is removed from the equation, which leads to a partitioning without the influence of data transfers. In the second step, this overhead is added, resulting in a more practical partitioning. By using the two-step modeling, it is possible to evaluate the impact of data transfers on the partitioning. Besides, data-transfer optimizations [56, 65, 79] and integrated CPU+GPU hardware designs (like Intel Sandybridge [51] and AMD APUs [4]) can hide or eliminate this overhead, therefore modeling the kernel execution time and the data-transfer time in two phases makes our method applicable to more applications and hardware platforms.

In the first step, we introduce two quantities W and P to evaluate the kernel execution time. W (the total workload size) quantifies how much work has to be done. Due to the workload imbalance, using the problem size (1D) to evaluate W is not sufficient. We define W as a 2D quantity by summing up the workloads of all the data points. P (processing throughput), defined as amount of workload processed per second, evaluates the hardware capability of a processor, indicating how fast the processor performs the application workload. As a result, the kernel execution time is given by W/P. Substituting W_G, P_G, W_C, and P_C into the first-step target ($T_G = T_C$), we have

$$\frac{W_G}{W_C} = \frac{P_G}{P_C} \tag{4.1}$$

Equation 4.1 indicates that the workload partitioning into W_G and W_C is affected by the relative hardware capability (P_G/P_C) between the two processors .

In the second step, we further quantify the data-transfer overhead. We use two more quantities O (data-transfer size) and Q (data-transfer bandwidth, measured in bytes per second), and we have $T_D = O/Q$. Q is determined by the bandwidth of the physical connection (e.g., PCIe) that connects the GPU to the CPU. We ignore the latency (smaller than 100 μs in our measurement) for setting up the data transfer, as it has negligible performance impact. Adding the definition of T_D into the second-step target ($T_G + T_D = T_C$) and performing equation transformations, we get

$$\frac{W_G}{W_C} = \frac{P_G}{P_C} \times \frac{1}{1 + (O/W_G) \times (P_G/Q)} \tag{4.2}$$

[2]In this chapter, we use the subscript G, C, and D to denote the GPU, the CPU, and the data transfer, respectively.

Equation 4.2 shows the impact of data transfers on the partitioning. Let F be the impact factor, $F = \frac{1}{1+(O/W_G)\times(P_G/Q)}$. As F is smaller than 1, W_G/W_C is decreased to compensate for the data-transfer penalty, resulting in a new partitioning with a part of the GPU workload moved to the CPU. Equation 4.2 specifies our final partitioning model.

4.2.4 The Prediction Method

Let β be the fraction of data points assigned to the GPU (β shows the partitioning point). Finding the optimal partitioning point is equivalent to solving β from Equation 4.2. There are two types of terms in Equation 4.2: (1) the partitioning-independent terms (i.e., the β-independent terms), P_G/P_C and P_G/Q; and (2) the partitioning-dependent terms (i.e., the β-dependent terms), W_G/W_C and O/W_G. To solve β from Equation 4.2, we need to estimate the β-independent terms, and use β to express the β-dependent terms.

We use profiling to estimate the β-independent terms: P_G/P_C (the relative hardware capability) and P_G/Q (the ratio of GPU throughput to data-transfer bandwidth). As the two ratios depend not only on the processors' hardware characteristics, but further on the application and the problem size that the processors perform, profiling the processors in the (application, problem size) context leads to a more realistic estimation. In addition, the use of profiling efficiently tackles the platform and workload diversities, making the prediction method adaptive to changes in platforms, applications, and problem sizes.

Our profiling is based on the *partial execution* of the target workload, i.e., we use a reduced workload to expose the global behavior of the application. As imbalanced applications are usually iterative based, we can limit the workload of every data point to a height v ($v_{min} \leq v \leq bh$)), resulting in a partial workload, noted as V, for profiling (see Figure 4.5). The choice of v is a trade-off between profiling accuracy and cost. In general, profiling a partial workload is much quicker than profiling a full workload. Moreover, profiling results can be reused for multiple workloads with the same problem size but different shapes, so the profiling becomes a one-time effort, and the overall prediction speed is further increased. We execute the V workload on the GPU and the CPU separately, record the execution time T_G^V and T_C^V, and obtain the following estimation:

$$\frac{P_G}{P_C} \approx \frac{W^V/T_G^V}{W^V/T_C^V} = R_{GC}, \quad \frac{P_G}{Q} \approx \frac{W^V/T_G^V}{O^V/T_D^V} = \frac{W^V}{O^V} \times R_{GD} \tag{4.3}$$

W^V is calculated according to the definition of W ($W^V = \sum_{i=0}^{n-1} v = n \times v$). O^V is obtained from the application, which is the full data-transfer size as we profile the whole problem size. Thus, by measuring the two time ratios, T_C^V/T_G^V (noted as R_{GC}) and T_D^V/T_G^V (noted as R_{GD}), we can estimate P_G/P_C and P_G/Q.

Next, we find the expressions for the β-dependent terms: W_G/W_C and O/W_G. We express the three quantities, W_G, W_C, and O, separately. For the data-transfer size (O),

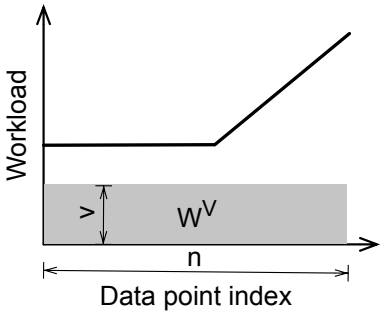


Figure 4.5: The partial workload V, used in profiling.

depending on the specific application, O can be proportional to the data points assigned to the GPU (thus $O = O^V \times \beta$), or fixed no matter how many data points the GPU gets (thus $O = O^V$). In the latter case, O becomes β independent, reducing the complexity of the problem. For the workload size of the GPU partition (W_G) and the CPU partition (W_C), because we define W as the sum of the workloads of all the data points, both of them can be determined by calculating the area (A) of the partitions (with the use of the workload model parameters, see Figure 4.4). Based on the position of the partitioning point, there are two cases of partitioning, and the expressions are accordingly different (see Figure 4.6). Therefore, we can express W_G and W_C by using the known parameters (n, bh, pw, ph, s, and α) captured by the workload model and the only unknown parameter β.

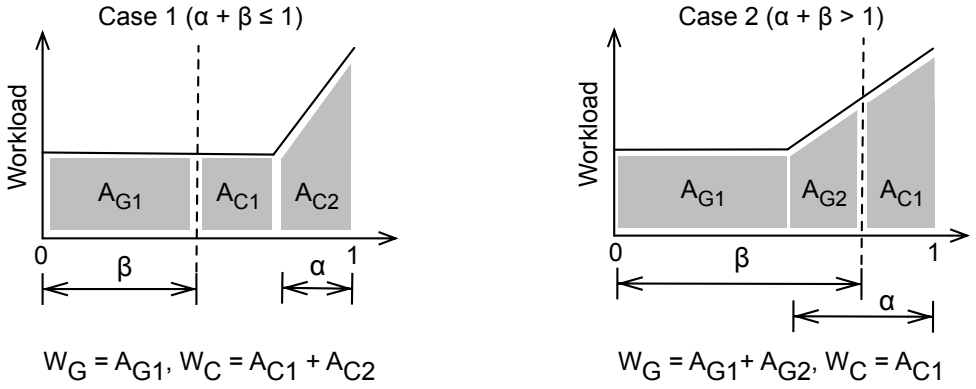


Figure 4.6: Two cases of partitioning. In Case 1, the partitioning point lies in the flat part. In Case 2, the partitioning point lies in the peak part.

Finally, by substituting the estimations of the β-independent terms and the expressions of the β-dependent terms into Equation 4.2, we get:

$$\frac{W_G}{W_C} = R_{GC} \times \frac{1}{1 + (\beta \times n \times v \times R_{GD})/W_G} \qquad \text{if } O = O^V \times \beta \tag{4.4a}$$

$$\frac{W_G}{W_C} = R_{GC} \times \frac{1}{1 + (n \times v \times R_{GD})/W_G} \qquad \text{if } O = O^V \tag{4.4b}$$

Because only β is unknown, by solving the equation we can get β. We implement the β solvers[3] of the two equations into the predictor. Depending on the application's data-transfer type, the predictor chooses to solve Equation 4.4a or 4.4b. As there are two cases of partitioning, and it is not possible to determine beforehand whether the partitioning lies in Case 1 or Case 2, the predictor calculates for both cases, and chooses the only one β that falls into the valid range (either $\alpha + \beta_1 \leq 1$ for Case 1 or $\alpha + \beta_2 > 1$ for Case 2). Using the predictor, we obtain the optimal partitioning point in O(1) time.

In summary, enabled by the use of the workload model and the profiling of the partial workload, our prediction method derives the optimal partitioning with a fairly low cost, and efficiently tackles the platform and workload diversities.

4.3 Experimental Evaluation

In this section, we use synthetic benchmarks to evaluate our prediction method. We focus on three important aspects: the prediction quality, the partitioning effectiveness, and the adaptiveness of the method. We further apply our method to a real-life application as a case study in Section 4.4.

4.3.1 Experimental Setup

Test Workloads: we develop a set of synthetic benchmarks to generate test workloads. These benchmarks are based on iterative vector updates. In each iteration, each data point i performs $A[i] \leftarrow A[i] + B$, where B is computed using 100 floating-point operations, sufficient for timing reliability. The final value of each data point is gathered at the host; thus, the GPU data transfer size is proportional to the partitioning β (we choose to examine the more complex case). To make the workload imbalanced, we vary the number of iterations per data point. The process is controlled at code level by setting the workload model parameters (see Table 4.1). Varying the values of these parameters, we get a set of benchmarks that generates a set of workloads with different amount of computations and memory operations (presented in different workload shapes).

Table 4.1: The values of each workload model parameter.

n	10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 10^8
α	$(10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}) \times j$, $(j = 1, 2, ..., 9)$, 1.0
pw_{min}	10
bh	500
ph	1000, 2000, 3000, 4000, 5000

[3] In this work, we use "β solver" to refer to the solution to the equation.

Table 4.1 summarizes the values we use: 7 different problem sizes (n), ranging from 10^2 (2KB, within the CPU cache) to 10^8 (2GB, around the GPU memory size). For each n, we vary the α value to change the peak width (pw). The minimum pw is fixed at 10 to make the peak part have at least 10 data points. The lager n is, the more α values are used, and the more workloads are generated (e.g., for a problem size of 10^2, the minimum α is 10^{-1}; for a problem size of 10^8, the minimum α is 10^{-7}). The maximum α is set at 1.0 for every n, case in which all the data points form a peak (i.e., there is no flat part in the workload). In the workload dimension, we keep the base height (bh) at 500 and change the peak height (ph) from 1000 to 5000, distinguishing the peak part from the flat part at various extents. The slope (s) of the peak part is also varied based on the values of α and ph. Figure 4.7 illustrates 6 example workload shapes. In total, we generate 1295 different workloads with different shapes (the number of workloads per problem size is shown in Table 4.3).

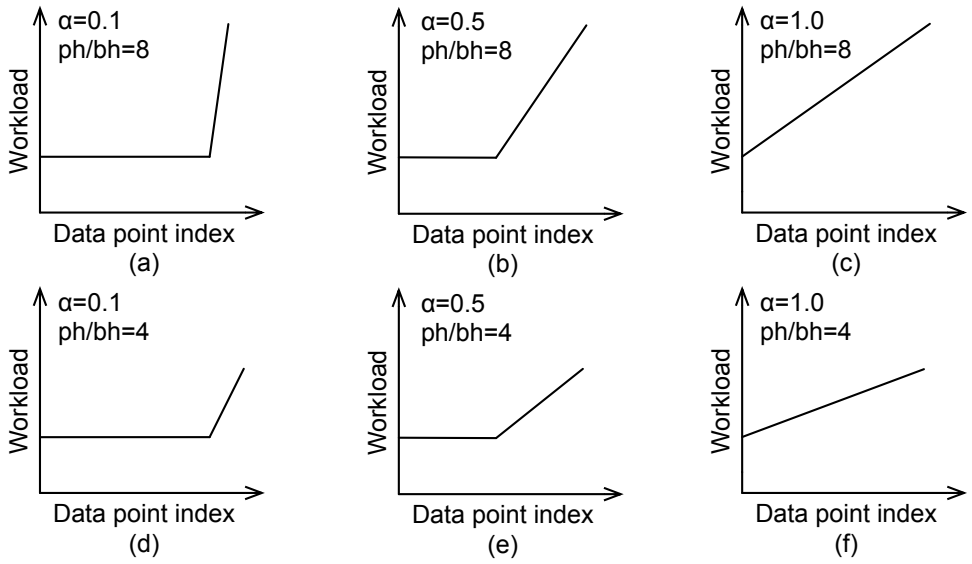


Figure 4.7: Examples of different workload shapes.

Heterogeneous platforms: we select three heterogeneous platforms for our experiments. The first platform (S1) integrates a dual-socket Intel six-core CPU (E5645, hyper-threading enabled) and an Nvidia Tesla C2050 GPU. The second platform (S2) consists of the same CPU as S1 and an Nvidia Quadro 600 GPU. The third platform (S3) has the same GPU as S1 and a dual-socket Intel quad-core CPU (E5620, hyper-threading enabled). Table 4.2 summarizes the hardware information. We use OpenCL to parallelize the code. The device-side compiler is Intel OpenCL SDK 2012 and NVIDIA OpenCL SDK 4.2, respectively. The host-side compiler is GCC 4.4.3 with -O3 option.

Table 4.2: The details of the heterogeneous platforms used in experiments.

Platform	S1		S2		S3	
Device	CPU	GPU	CPU	GPU	CPU	GPU
Processor	2×Xeon E5645	Tesla C2050	Same as S1	Quadro 600	2×Xeon E5620	Same as S1
Frequency (GHz)	2.4	1.15		1.28	2.4	
#Cores	24 (HT enabled)	448		96	16 (HT enabled)	
Peak GFLOPS (SP/DP)	460.8/230.4	1030.4/515.2		245.8/122.9	307.2/153.6	
Memory Capacity (GB)	24	3		1	24	
Peak Memory BW (GB/s)	64	144		25.6	51.2	

4.3.2 Prediction Quality

Our first goal is to validate prediction quality, i.e., to check if the prediction method finds the optimal partitioning. We use the partitioning obtained through auto-tuning as an estimate of the optimal partitioning that meets $T_G + T_D = T_C$. In practice, a threshold δ is applied to control auto-tuning. When $|T_G + T_D - T_C| \leq \delta$, the auto-tuning is stopped, and the resulting partitioning is considered as the optimal one.

We measure prediction quality by calculating the execution-time difference D between the predicted partitioning and the optimal partitioning. D shows how the predicted partitioning approximates the optimal partitioning in terms of performance. We use the difference, rather than the relative error, to assess prediction quality. Because the auto-tuning is controlled by a threshold, a certain degree of execution-time variation between the predicted partitioning and the auto-tuned partitioning is acceptable. Thus, we apply the same threshold to D: if $|D| \leq \delta$, we consider that the predicted partitioning is optimal. In this way, we distinguish high-quality predictions from outliers. The threshold δ we use is 1, 1, 2, 3, 5, 15, 60 (ms) for problem sizes from 10^2 to 10^8, respectively. We choose these values, taking the execution time and the timing measurement error[4] into account.

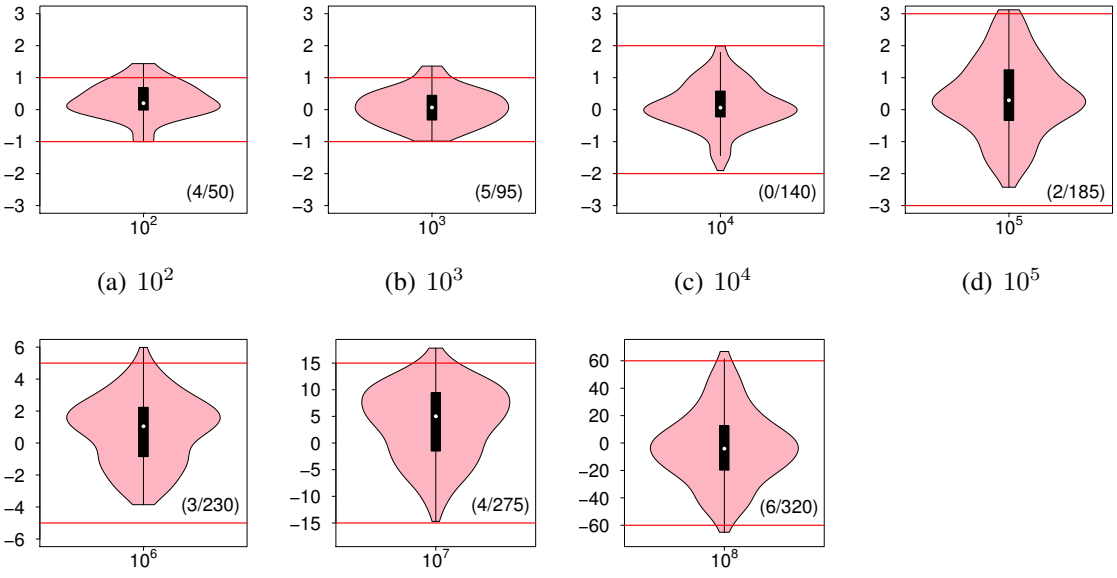
Table 4.3: The number of workloads and the hardware profiling results (R_{GC}, R_{GD}), per problem size.

n	10^2	10^3	10^4	10^5	10^6	10^7	10^8
#workloads	50	95	140	185	230	275	320
R_{GC}	0.80	3.76	20.09	30.09	30.47	30.60	30.61
R_{GD}	0.42	0.42	0.36	0.17	0.10	0.09	0.08

The experiment is performed on the S1 platform. To predict β, we measure R_{GC} and R_{GD} (see Equation 4.3) by profiling the partial workload V. We set $v = bh$, so we can

[4]The timing measurement error is within the threshold we choose. Each timing test is repeated 50 times. The unstable results in the warm-up phase are discarded, and then the median value in the stable results is selected as the final timing result.

see the maximum profiling cost. The obtained R_{GC} and R_{GD} are presented in Table 4.3. Their values are different for different n, proving that profiling in the (application, problem size) context leads to more realistic estimations. The profiling results are reused for all the workloads with the same n, so the profiling cost is amortized (0.9%–9% of the total execution time of all the workloads). We also find that the choice of v_{min} is platform dependent. Lowering v beyond a certain height, the prediction quality will not be acceptable. In this experiment, v_{min} is empirically set to 100.



(a) 10^2 (b) 10^3 (c) 10^4 (d) 10^5

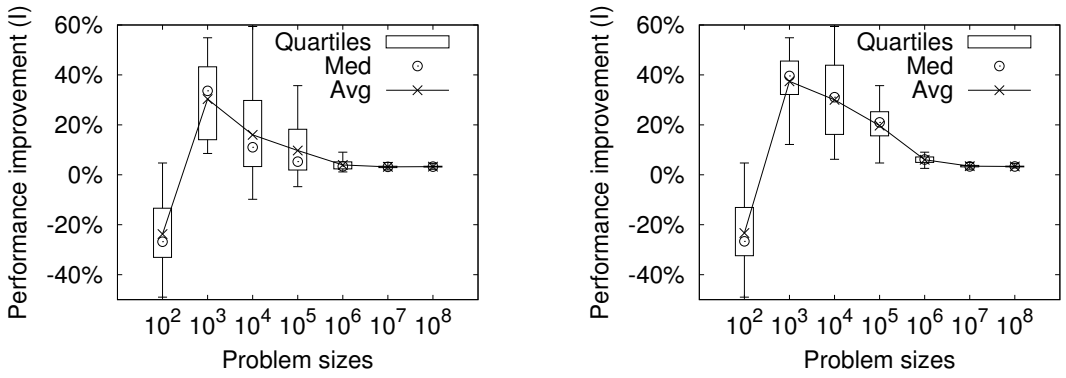
(e) 10^6 (f) 10^7 (g) 10^8

Figure 4.8: The prediction-quality results, in violin plot, per problem size. The vertical (y) axis represents the execution-time difference D (in ms). The horizontal lines represent the threshold δ. The violin shape represents the distribution of D values. The numbers between parenthesis show #outliers vs. #total test workloads. (The y-axes of the seven figures have different scales.)

By using the predictor, we obtain β for each workload. By contrast, the auto tuning needs on average 3–8 rounds of binary search (the execution of the full workload) to get the optimal partitioning. The execution-time difference D per problem size is summarized in Figure 4.8. We see that the prediction obtains the optimal partitioning in 98% of the total tests (1271 out of 1295 tests). For most problem sizes, the distributions of D values are concentrated in the middle of the thresholds, further proving the prediction's high quality. In summary, our method is able to predict the optimal partitioning for imbalanced workloads with very small (0.9%–9%) profiling cost.

4.3.3 Partitioning Effectiveness

To evaluate partitioning effectiveness means to understand if workload partitioning on heterogeneous platforms is worth doing, i.e., if our partitioning solution outperforms the homogeneous solution (we run the application only on the CPU and the GPU, and choose the one that performs better). We use I, the percentage of performance improvement, to evaluate the effectiveness (assuming that partitioning improves performance).



(a) the original results

(b) excluding the small-slope cases

Figure 4.9: The performance improvement, in box-and-whisker plot, per problem size: (a) the original results, (b) the results after excluding the small-slope cases.

We run the same 1295 workloads on S1. Figure 4.9(a) shows the performance improvement results. For larger problem sizes (10^6–10^8), the maximum I is smaller than 10%. As the GPU is much capable for the given workloads, and the data-transfer overhead is very small (see Table 4.3, $R_{GC} > 30$, $R_{GD} \leq 0.1$), the benefit of workload partitioning is limited. Our prediction reflects this by assigning more than 95% of the total data points to the GPU. For the smallest problem size (10^2), most workloads have negative I values (see the 3/4 quartile). Because the CPU has higher capability to process the workloads ($R_{GC} < 1.0$), using the GPU is a waste of resources, and there is no need for partitioning. The median problem sizes (10^3-10^5) benefit the most from partitioning, reaching up to 60% performance improvement. We also notice that the distributions of I values are very scattered in these three problem sizes, and there are also negative I values. This behavior is related to the peak slope s (see Figure 4.4). When the peak has a small slope, the workload imbalance is less prominent, making the homogeneous solution with no data-transfer overhead a better option. Based on our empirical study, the small-slope upper bound is set to 5. Excluding the small-slope cases for all the problem sizes, we show the new results in Figure 4.9(b). We see that for the problem sizes of 10^3-10^5, the average I values are increased (above 20%), and the distributions of I values are more concentrated and upward shifted (see the change of the minimums and the quartiles). By

contrast, the other problem sizes show no significant changes.

To summarize, partitioning is an effective method for imbalanced workloads, as it leads to better platform utilization than the homogeneous solution. The more imbalanced the workload is, the more performance gain is obtained. There are two cases when using a single processor is better. First, when the dataset has not enough parallelism, using only the CPU (no partitioning) is sufficient. Second, when the GPU has much higher capability to process the workload and the data-transfer overhead is negligible, workload partitioning has limited performance gain. We are able to know this beforehand from prediction ($\beta > 95\%$), and we can use only the GPU for execution.

4.3.4 Adaptiveness of the method

Another important aspect is the adaptiveness of the method. The above experiments have shown that our method is able to respond to workload and problem size changes. To evaluate its adaptiveness to platform changes, we perform two more experiments.

First, we use the S2 platform (see Table 4.2) that has the same CPU as S1 but a less powerful GPU (both GPUs have the same Fermi architecture). The number of cores, peak GFLOPS, memory capacity, and bandwidth are reduced compared to that in S1. We run the same 1295 workloads on the new platform. As the GPU is changed, we re-profile the GPU, and recalculate R_{GC} and R_{GD} for prediction (the CPU profiling results are reused). We see that the resulting prediction maintains high quality and reacts to the GPU change by shifting the partitioning point one direction or another.

Table 4.4: The change of the partitioning point when using new platforms.

Platform	Problem size	Hardware Capability	New Partitioning
S2	10^2–10^3	new GPU > old GPU	right shifted
	10^4–10^7	new GPU < old GPU	left shifted
	10^8	new GPU < old GPU	left shifted till it fits the GPU memory
S3	10^2–10^8	new CPU < old CPU	right shifted

Table 4.4 summarizes the change of the partitioning point. For smaller problem sizes (10^2–10^3), the new GPU is more efficient at processing their workloads than the old GPU. As a result, the partitioning point is right shifted (the new GPU gets more work), and the performance is improved by 12%. For larger problem sizes (10^4–10^7), the new GPU has much lower capability than the old one, thus the partitioning point is largely left shifted (the CPU gets more work), and the performance is decreased by up to 4 times. For the largest problem size (10^8), the calculated GPU partition exceeds the new GPU's memory capacity. For now, the prediction chooses a smaller partition that fits the memory, but lowers the performance. In Chapter 5, we solve this limitation by using platforms with multiple GPUs.

Second, we use the S3 platform, where the GPU is the same and the CPU is changed to a lower configuration (dual quad-core, lower peak GFLOPS and memory bandwidth). We repeat the experiment and re-profile only the new CPU. Again the prediction reacts to the CPU change by moving more work onto the GPU. We also find that the performance difference between S1 and S3 is quite small compared to that between the two CPUs, indicating that the partitioning compensates for the CPU degradation.

To sum up, our prediction method is able to tackle the platform diversity as well. It responds to hardware changes by re-profiling the new hardware and resizing the partitions accordingly.

4.3.5 The Effect of Compilers

We also investigate how our method responds to compiler performance. As Nvidia and Intel release new versions of their compilers quite often, we make an experiment to test the impact of these changes. We update to Intel SDK 2013 and Nvidia SDK 5.5[5], and re-run the experiment on S1. We find that the Intel compiler improves application performance (on the CPU), while the new Nvidia compiler (the OpenCL part) does not (on the GPU). Our method responds to this change by re-profiling and re-predicting the partitioning. As a result, the CPU gets more work, and the new partitioning achieves up to 15% performance improvement.

Thus, we point out that a pure hardware modeling, or a hardware modeling combined with an (application, problem size) modeling, may lead to a sub-optimal partitioning when ignoring the effect of compilers. As updating or altering compilers can affect application performance on the same hardware platform, profiling is essential to achieve the best performing partitioning.

4.4 A Real-world Case Study

In this section, we use acoustic ray tracing introduced in Chapter 3 as an imbalanced application case study, and show how our method is applied to real-world applications.

4.4.1 The Application

Acoustic ray tracing [6] is developed to calculate, in real time, the propagation of aircraft flyover noise by closely simulating the changes in the ray trajectory as it moves through different media. Figure 4.10 illustrates the propagation of all the rays launched by the aircraft at different angles.

[5] At the time when the experiment was performed (January 2014), these were the latest compilers.

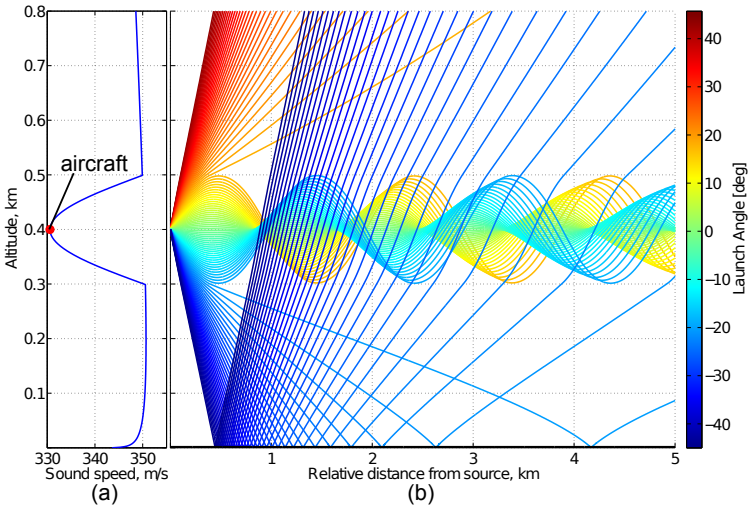


Figure 4.10: Ray propagation in an $altitue \times distance$ grid. (a) Sound speed profile. There is a headwind (at 0.3-0.5 km altitude) blowing against the aircraft (the red dot at 0.4 km altitude). (b) The propagation profile for all the sound rays emitted by the aircraft.

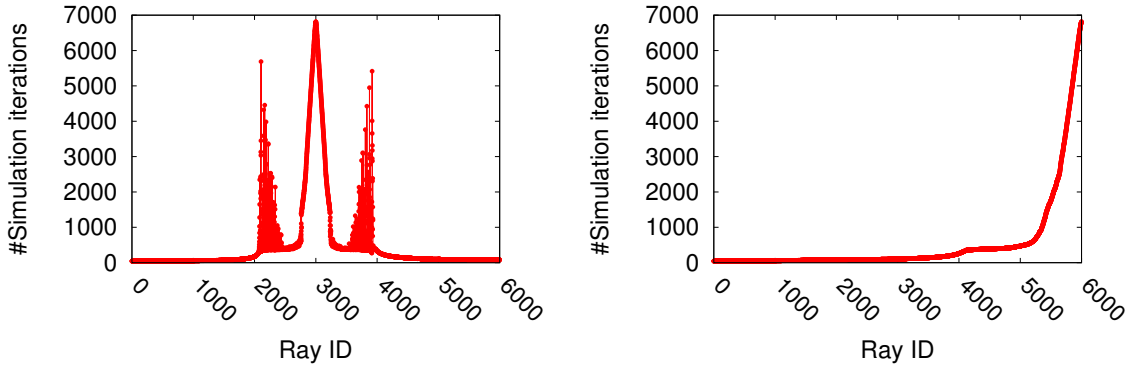
The ray propagation path is determined by a time-based simulation, where the location and orientation of a ray is calculated in every time step, and saved for later auralization. In the implementation, there are two nested loops: the outer loop traverses all the rays, and the inner loop computes the ray propagation. In the inner loop, the variable time step technique is applied to control the simulation resolution. When a ray meets certain location and orientation conditions, the time step of the ray is dynamically refined to obtain high fidelity (i.e., its propagation is more interesting for the users). It is this use of the variable time step that leads to an imbalanced workload. All the rays are independent from each other and simulated in parallel, but the workload of each ray is dynamically determined at runtime.

By changing the flyover environment (e.g., the headwind speed, the aircraft height), we obtain 10 real imbalanced workloads with different workload shapes. Each workload has 6000 simulated rays ($n = 6000$), and the workload of a ray is measured by the number of simulation iterations. Figure 4.11 shows one such workload.

4.4.2 The Partitioning Process

To partition the acoustic ray tracing workload, we need to do three things: model the workload, profile the platform, and perform the prediction.

First, workload modeling extracts the workload model from the original workload. It consists of two steps: workload sorting and workload model approximation. The sorting reshapes the original workload into a peak-shaped workload (see Figure 4.11(b)), making



(a) the original workload (b) after sorting

Figure 4.11: One workload of acoustic ray tracing: (a) the original workload, (b) the workload after sorting. Each ray is identified by its ray ID starting from 0 to 5999, which corresponds to its launching angle ranging from -45 to 45 degree with respect to the horizontal. After the sorting, each ray has a new ID, and we record the mapping between the original and new IDs for de-sorting.

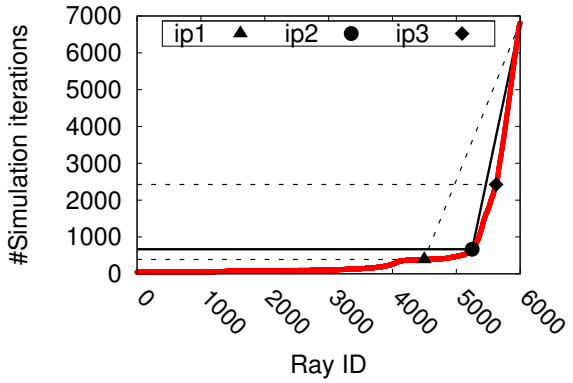


Figure 4.12: For the workload depicted in Figure 4.11(b), the three workload-model approximations shown by the straight solid lines and dashed lines. $ip1$, $ip2$, $ip3$ are the three last-searched points. In this example, $ip3$ has the largest Δs, but $ip2$ leads to the closest workload approximation, and thus to the best performing partitioning.

the model approximation easier. The model approximation finds out the inflection point ip between the flat part and the peak part of the sorted workload, and obtains the workload model. It uses a binary search to get the ip that has the largest slope difference Δs (Δs is calculated by connecting ip with the 0-th point and the 5999-th point, and getting the slope difference between the two lines). The three last-searched points are recorded as candidate ip, i.e., we approximate the workload model per ip, predict the partitioning, and choose the best performing one. Figure 4.12 shows the three workload-model approximations for the workload depicted in Figure 4.11(b). Next, we use partial profiling

to profile the application on the given platform. The obtained profiling results (R_{GC} and R_{GD}) are used for all the 10 workloads as they have the same problem size. Finally, the generated workload models and the profiling results are combined in the predictor to get the partitioning. Note that a de-sorting is applied at the end of execution to assemble the correct simulation output.

Costs vs. Performance: There is a small cost to model the workload and profile the platform. Depending on the user scenarios, the cost will vary. For example, if the best application performance is the target, we can use total sorting, which potentially guarantees the best performance and takes on average 6% (sorting and de-sorting) of the application execution time. If the performance requirement can be relaxed, a partial sorting [34] or a sampling combined with sorting can be used. This decreases the application performance by 24% (due to a less accurate workload model), but reduces the modeling cost to less than 2% of the application execution time. For workload model approximation, examining three ip candidates leads to three times of runs of application. Choosing only one of them may result in a suboptimal partitioning, but reduces the modeling cost to only the ip searching time (0.2% of application execution time). For profiling the platform, the cost (10% of the application execution time) can be amortized by all the tested workloads, thus the more workloads we run, the less profiling cost per workload we get. The real application scenario is even simpler: different people will hear the same flyover noise simulation, leading to a survey of its annoyance factor. In this scenario, the same flyover environment is set, and the same workload is executed repetitively. Thus, the modeling and the profiling become a one-time effort, as the users can use them just once for prediction, and reuse the prediction by fetching it from the repository.

4.4.3 The Partitioning Results

We evaluate the prediction quality and the partitioning effectiveness as we have done for synthetic workloads. We perform the experiment on S1 (see Table 4.2). Figure 4.13 shows the experimental results (note that because the application has real-time requirement for auralization, the execution time is in ms scale). By comparing the execution time of the predicted partitioning and the best partitioning obtained through auto-tuning, we see that, for 9 out of 10 workloads, the execution-time differences are within the threshold ($|D| \leq \delta$, δ is set to 3 ms), with the average D at 1.2 ms. The exception is W3, which actually has a stair-shaped workload. As the workload modeling approximates W3 with a wider peak, this coarse approximation leads to less accurate prediction and degraded performance. This issue can be addressed by refining our workload model to include stair-shaped workloads. Comparing the execution time of the predicted partitioning and the single CPU/GPU solution, we see that the partitioning largely improves application performance, with the average performance improvement I at 51% (the maximum I is

85% in W6). We notice that a workload with a larger peak slope (a more imbalanced workload) gets higher performance improvement. Only W4 has limited performance gain. This is mainly because it falls into the small-slope case, in which most of the workload is assigned to the GPU. The predicted partitioning actually indicates this, so we can decide to use only the GPU for W4 without applying the partitioning.

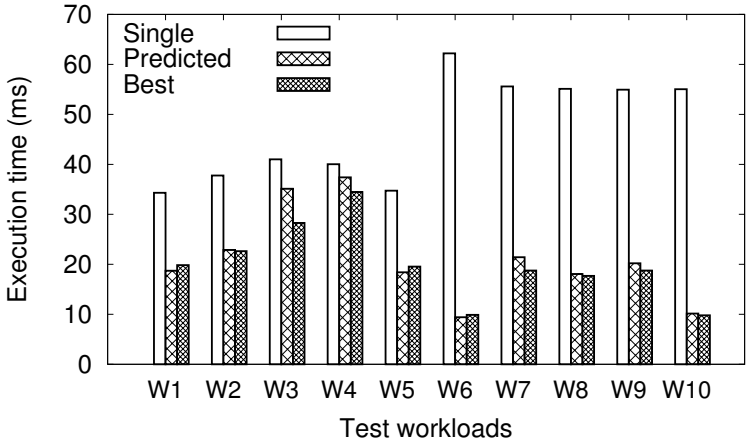


Figure 4.13: The execution time per real imbalanced workload. W1-W10 denote the 10 workloads. "Single" represents using only the GPU or the CPU (the one with smaller execution time is chosen). "Predicted" represents the predicted partitioning. "Best" represents the best partitioning obtained through auto-tuning.

Summary: In practice, to achieve the best performance for real-world imbalanced applications on heterogeneous platforms, we should apply the following recipe: (1) model the workload, profile the platform, and perform the prediction, (2) decide for partitioning or not based on (1), and if needed, (3) apply the predicted partitioning. The empirical study shows that the decision for partitioning or not is made according to the application problem size and the prediction results. If the application problem size is too small (not enough parallelism), we should use only the CPU. If the prediction decides to run most of the workload on the GPU, we can directly use the GPU, and there is no apparently performance lose.

4.5 Related Work

Workload partitioning for heterogeneous platforms has been well studied in recent years. Multiple partitioning approaches have been developed aiming to efficiently utilize all the computing resources of the heterogeneous platform.

Merge [76] and the work of Tomov et al. [125] determine the right workload partitioning according to the performance estimate provided by the programmers. These solutions have limited applicability as they largely rely on manual work. Our work in Chapter 3 has demonstrated the use of auto-tuning for workload partitioning. Because auto-tuning can be time-consuming and occupies the target platform unnecessarily, the prediction-based method presented in this chapter is a mandatory improvement for our workload partitioning approach.

Grewe et al. [37] adopted a machine learning approach to predict the workload partitioning. They built a predictive model based on static program features extracted from the code. Kofler et al. [67] improved this approach by adding dynamic program features (i.e., problem-size related features) into the model. Compared to our approach, both solutions are not adaptive to hardware and software changes. When either changes, a rerun of the whole training phase is needed to rebuild the predictive model, and this training overhead cannot be ignored.

Luk et al. [78] proposed an adaptive approach, Qilin, to map computations to heterogeneous platforms. Qilin uses a large amount of profiling data and curve-fitting to derive the partitioning for the same program with a new input. Our prediction uses a partial workload for profiling, which largely reduces the profiling cost and still generates an accurate prediction. Gharaibeh et al. [33] built a performance model to study the partitioning of graph workloads. They also introduced hardware-capability related metrics to estimate graph processing performance. However, they assumed the metrics are constant independent of actual graph workloads, leading to lower accuracy for some cases. Instead, we use a more realistic estimation by measuring the metrics in the (application, problem size) context. Yang et al. [134] predicted application performance on large-scale systems by using relative performance between the target and reference systems. Our partial workload profiling relates to their approach as we both adopt a black-box style performance observation from a very short run of the application. While their prediction is used for large-scale system selection, we focus on workload partitioning over heterogeneous platforms.

On the other hand, studies on dynamic workload scheduling [8, 14, 80, 103, 108] aim to achieve load balancing in heterogeneous systems. Workload scheduling is able to respond to performance variability at runtime, but it may lead to high scheduling and communication overheads depending on the applications, the hardware, and the scheduling algorithms. In our work, we find that workload partitioning is more efficient at accelerating imbalanced workloads on CPU+GPU platforms, as workload scheduling may generate successive data communication between the CPU and the GPU.

All the above mentioned work only consider the problem-size dimension for partitioning. Our approach takes the workload dimension and workload characteristics into consideration, and generates accurate, efficient partitioning solutions.

4.6 Summary

Workload partitioning is essential for efficiently utilizing heterogeneous platforms. In this chapter, we propose a prediction-based partitioning method to match imbalanced workloads with CPU+GPU heterogeneous platforms. Our method predicts the optimal partitioning for imbalanced workloads, leading to the best performance on the platform. We evaluate our method in terms of prediction accuracy, overall performance gain, flexibility and adaptivity. Through our experiments on both synthetic workloads and a real-world case study, we show evidence that our method has high prediction quality (above 90% of all the 1395 tests get accurate prediction), increases application performance by using both processors (up to 85% performance improvement over a single processor execution), and maintains a very low partitioning cost. We also show that the method is adaptive to application, problem size, hardware, and software changes, obtaining the optimal partitioning for any given (workload, platform) pair. We conclude that model-based prediction of workload partitioning optimizes the partitioning process, and is both feasible and useful for accelerating imbalanced applications on heterogeneous platforms.

Chapter 5

Generalizing Workload Partitioning: A Systematic Approach

In Chapter 4, we have proposed a model-based prediction method to optimize the workload partitioning process for imbalanced applications. In this chapter, we propose a systematic workload partitioning approach, based on the model-based method, to improve the performance for both balanced and imbalanced applications, for applications with different datasets and execution scenarios, and for platforms with multiple accelerators.

People tend to choose GPUs when accelerating balanced (data parallel) applications, as balanced applications have regular, uniform workloads, and GPUs are tagged as massive-parallel and high-throughput machines.

However, GPUs are not always the best-performing solution on their own. One needs to take into account that GPUs are designed as accelerators and cannot work in a standalone mode: the host, which is a CPU, is needed to manage application execution. Modern CPUs have multiple cores and wide vector (SIMD) units, offering thread- and data-level parallelism. Actually, the performance gap between GPUs and CPUs is not as large as it appears (e.g., on average 2.5× in [73]) after applying appropriate optimizations for both. More importantly, this gap is most likely to vary as applications and datasets vary [17, 24, 124]. In other words, the host CPU is essentially a free hardware resource, and can be utilized, together with/instead of the GPU, to improve the overall performance of a workload [33, 44, 122, 128, 133].

Additionally, a data-transfer bottleneck appears as data to be consumed and produced by the GPU must be transferred from and to the host CPU (the GPU and the host CPU have separate memory spaces). This transfer is performed through a physical connection, typically a PCI Express bus. Because the GPU and CPU throughputs are higher and increasing much faster than the bandwidth of the physical connection, this data transfer tends to become a performance killer, cancelling out any gain obtained on the GPU [36].

In this situation, using only the host CPU is more often than expected the best-performing option.

These observations indicate that it is important to reconsider the acceleration of balanced applications. Instead of using only the GPU, we should exploit the native heterogeneity of the GPU platform, i.e., the GPU *plus* the host processor (the CPU). To obtain the best performance on such a heterogeneous platform, we need to find the right hardware configuration and the optimal workload partitioning. This partitioning is a function of the hardware capabilities as well as the application and the dataset to be used.

Finding such a workload partitioning remains a challenge. Firstly, the *heterogeneity* of the platform requires the partitioning to fully consider the processors' differences in processing capability, hardware architecture, and memory model. For example, GPUs usually offer higher processing throughput than CPUs, but have separate memory spaces and low data communication bandwidth to the host memory. Secondly, the *diversity* of platforms, applications, and datasets increases the partitioning complexity. The diversity lies in the fact that heterogeneous platforms exist in different configurations and forms (e.g., multi-core CPUs with accelerators, embedded systems, MPSoC, etc.), and that data parallel applications (with different datasets) present various kinds of performance behavior even on the same platform. Finally, from the programmer's viewpoint, it is challenging to obtain, with *limited time and effort*, the optimal partitioning that leads to the best performance for a given heterogeneous platform, application, and dataset.

Several studies [37, 67, 76, 78] have proposed partitioning solutions for data parallel applications. None of them fully tackle the above three challenges: they are either not able to fit new platforms, different applications and datasets, or depend on expensive learning and/or the programmer expertise to derive the optimal partitioning.

In this work, we propose a systematic approach to solve the partitioning problem by determining (1) the right hardware configuration and, when needed, (2) the optimal workload partitioning. We mainly focus on common single-kernel data parallel applications, as they are the building blocks of multiple-kernel data parallel applications. We start with the most popular hardware setting, a CPU+GPU heterogeneous platform, and we have three possible hardware configurations: *Only-CPU* (run the application on the CPU), *Only-GPU* (run the application on the GPU), *CPU+GPU* (run the application in parallel, on both the CPU and the GPU).

To characterize the heterogeneity of the platform, we propose two metrics : (1) *the relative hardware capability*—the ratio of GPU throughput to CPU throughput, and (2) *the GPU computation to data transfer gap*—the ratio of GPU throughput to data-transfer bandwidth. We find that the two ratios are determining factors to the partitioning and, moreover, their values vary depending not only on the hardware processors but on the application and the dataset as well. To tackle the diversity issue, we estimate the two ratios in the presence of the application workload (i.e., the application and its dataset)

by using profiling. We generalize the partitioning model (first proposed in Chapter 4 for imbalanced applications), based on the two ratios, to predict the optimal partitioning for balanced applications. According to the prediction result, we finally determine if the partitioning is necessary and choose the corresponding hardware configuration.

To provide the users with the control of the partitioning overhead and accuracy, we propose different profiling options for different execution scenarios. We also extend the partitioning model to platforms with one CPU connected to multiple GPUs, further improving the applicability of our approach.

For heterogeneous computing, one needs to have parallel implementations for both CPUs and GPUs, and enable multi-device code for partitioned execution when necessary. If the code is not available yet, we recommend using OpenCL for parallelization, because it ensures the same parallelization structure across processors, and allows for code specialization for different processor families [109, 111, 112]. Although our approach supports any combination of programming models, this work will only include OpenCL examples.

We evaluate our approach on 12 applications with different problem sizes. Due to the determined optimal hardware configuration and partitioning, we see 1.2×–14.6× performance gain compared to a single-processor execution. The speedup variation is application and dataset dependent. If we choose a wrong hardware configuration ignoring the application and dataset used, up to 96% of the achievable performance is lost. In most test cases, the performance difference between our predicted partitioning and an oracle-based partitioning is within 10%.

Our main contributions are as follows: (1) we generalize a systematic approach to determine the right hardware configuration and the optimal workload partitioning for accelerating data parallel applications on heterogeneous platforms; (2) we propose two novel metrics (the relative hardware capability and the GPU computation to data transfer gap) that play a decisive role in determining the optimal partitioning; (3) we propose a profiling-based method (with multiple profiling options that allow users to tune the overhead-to-accuracy balance) to estimate the two metrics; (4) we show both theoretically and practically how our approach is applicable to balanced applications with different datasets and execution scenarios, and to platforms with different hardware mixes.

The rest of the chapter is organized as follows. Section 5.1 presents three case studies that illustrates the importance of using the right hardware configuration and workload partitioning, leading to Section 5.2 detailing the systematic approach. Section 5.3 presents an experimental evaluation of our approach, followed by a discussion in Section 5.4. We study related work in Section 5.5, and summarize this chapter in Section 5.6.

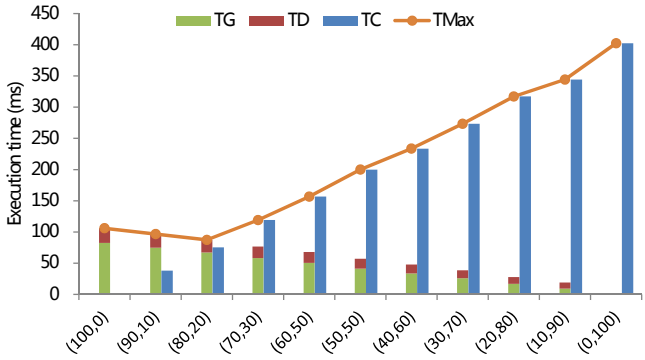
5.1 Partitioning and Performance: Three Case Studies

To show how partitioning influences application performance, we present three case studies: Matrix Multiplication (SGEMM), Separable Convolution (SConv), and Dot Product (SDOT), all extracted from the Nvidia OpenCL SDK.

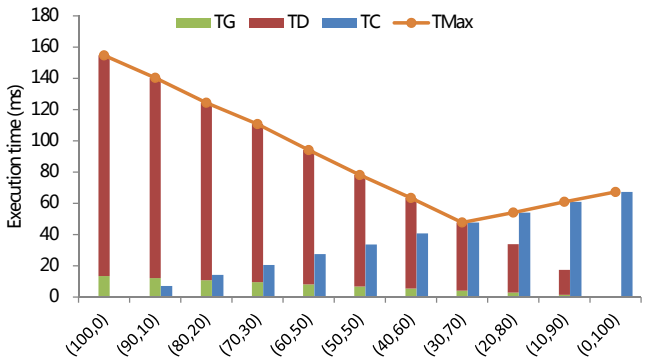
We perform a similar experiment for each case study: we partition the application to run on the CPU and the GPU, and vary the size of the partitions from assigning 100% work to the GPU (Only-GPU) to assigning 100% work to the CPU (Only-CPU) with a granularity of 10%. We measure the execution time of each processor and the overall time (the maximum time) after overlapping the executions of the two processors. For the GPU, as the data needs transferring, its execution time consists of two parts, the kernel-computation time and the data-transfer time, which are recorded separately. Figure 5.1 shows the execution time breakdown.

For SGEMM, as the GPU has higher throughput than the CPU (for 100% work, the GPU processes the kernel computation 5× faster than the CPU), and as the data-transfer time takes a small proportion (around 20%) of the whole GPU execution time, the best performance (the minimum T_{max}) is achieved with the configuration of 80% work on the GPU and 20% work on the CPU. For SConv, the GPU also has around 5× higher throughput, but its overall performance is diminished by the data-transfer overhead (the data transfer takes 11× more time than the GPU kernel computation). As a result, the best configuration is to make 70% work remain on the CPU. For SDOT, the impact of the data transfer on the GPU is even more significant: the kernel-computation time is almost invisible compared to the data-transfer time (up to 46× difference). This large data transfer cancels out any benefit the GPU contributes, making Only-CPU the best configuration.

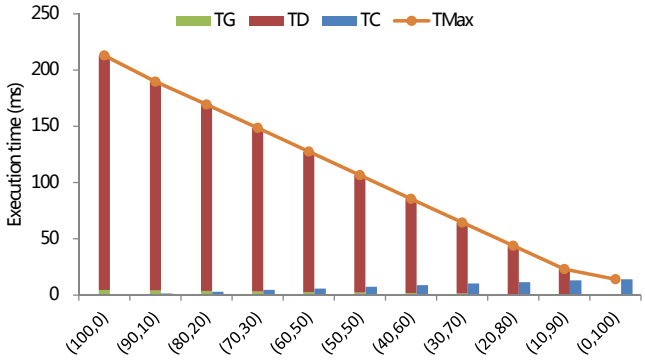
These three case studies demonstrate that the quality of the partitioning determines the application performance, and the optimal partitioning is a function of (1) *the relative hardware capability* (captured as the ratio of GPU throughput to CPU throughput), and (2) *the GPU computation to data transfer gap* (captured as the ratio of GPU throughput to data-transfer bandwidth). Moreover, the two ratios vary with the change of applications. For instance, SGEMM and SConv have similar relative hardware capabilities, but their GPU-computation-to-data-transfer gaps differ widely. SGEMM and SDOT are both linear algebra applications, but their two ratios are completely different. Even for the same application with different datasets or on different platforms, the ratios vary (shown in Section 5.3). Thus, we need to develop a systematic approach that captures the variations of the two ratios, and determines the optimal partitioning and hardware configuration accordingly.



(a) SGEMM



(b) SConv



(c) SDOT

Figure 5.1: The execution time breakdown with different hardware configurations. The tuple (X,Y) denotes the configuration of X% work on the GPU and Y% work on the CPU. Thus, the tuple (100,0) represents using only the GPU; the tuple (0,100) represents using only the CPU; any tuple in between represents using a mix of both. T_G, T_D, T_C, and T_{Max} denote the GPU kernel-computation time, the data-transfer time, the CPU kernel-computation time, and $max(T_G + T_D, T_C)$, respectively.

5.2 A Systematic Approach

Our systematic approach starts with a platform consisting of one CPU and one GPU, and assumes that CPU+GPU is the right hardware configuration. Then, we follow three main steps. First, *modeling the partitioning*: build a partitioning model which represents the execution of the partitioned workload on the CPU and the GPU. Next, *predicting the optimal partitioning*: calculate the optimal partitioning by solving the partitioning model. Finally, *making the decision in practice*: determine the right hardware configuration with practical consideration. Figure 5.2 depicts the whole approach, and Sections 5.2.1–5.2.3 describe each step in detail. The parameters used in our approach are summarized in Table 5.1. We apply the same partitioning principle as that used for imbalanced applications in Chapter 4, and we generalize this approach to balanced applications with different profiling options useful for different execution scenarios and to platforms with multiple identical or non-identical accelerators.

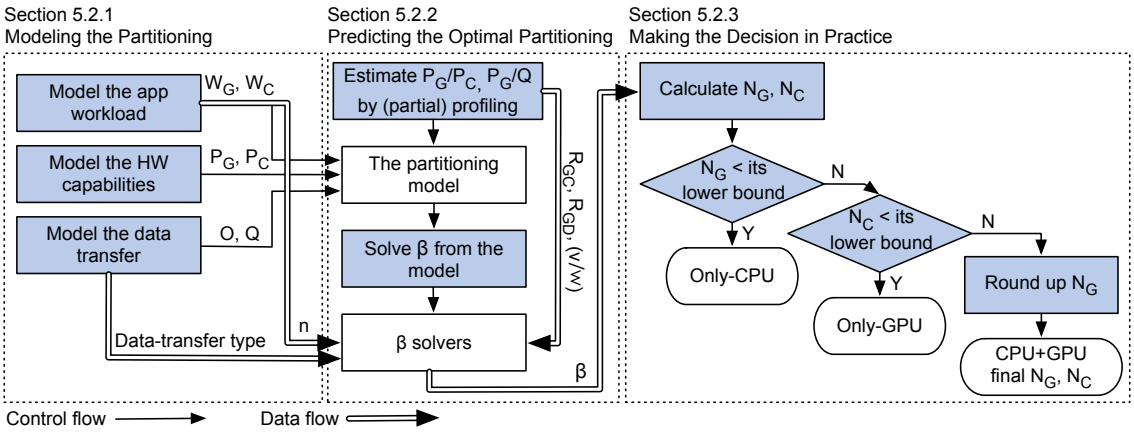


Figure 5.2: The systematic approach to determine the optimal partitioning and the right hardware configuration. It consists of three steps, and each step is labeled with the corresponding subsection that describes it.

5.2.1 Modeling the Partitioning

To model the partitioning means to model the execution overlap between the CPU and the GPU. As shown in Section 5.1, the optimal partitioning is the point where the perfect execution overlap is achieved: $T_G + T_D = T_C$. This equation defines the optimality, where we assume that the CPU manages the data transfer and processes its workload share in the same time, typically implemented by using different threads for data transfer and computation.

To model T_G and T_C, we introduce the total workload size (W) and the processing throughput (P). Assuming the application problem size is n, and the workload for each

Table 5.1: The parameters used in the approach.

n	The problem size, i.e., the number of independent data points in the application parallelization space
β	The fraction of data points assigned to the GPU, i.e., the partitioning point
w_i	The workload for a data point i in the problem space; for balanced applications, $w = w_i$
W	The total workload size, $W = \sum_{i=0}^{n-1} w_i$; for balanced applications, $W = n \times w$
W_G, W_C	The workload size of the GPU partition and the CPU partition, respectively
P_G, P_C	The processing throughput of the GPU and the CPU, respectively
O	The data-transfer size
Q	The data-transfer bandwidth
P_G/P_C	The relative hardware capability
P_G/Q	The computation to data transfer gap
W^V	The profiled workload size; if using full profiling, $W^V = W = n \times w$; if using partial profiling, $W^V = n \times v$ ($v < w$)
T_G^V, T_C^V, T_D^V	The profiled GPU kernel-computation time, the CPU kernel-computation time, the data-transfer time, respectively
R_{GC}	The time ratio computed as T_C^V/T_G^V
R_{GD}	The time ratio computed as T_D^V/T_G^V
N_G, N_C	The GPU partition size ($n \times \beta$), and the CPU partition size ($n \times (1-\beta)$), respectively

data point i in the problem space is w_i, we define the total workload size as $W = \sum_{i=0}^{n-1} w_i$ (i.e., the sum of the workload of all the data points). As we model balanced applications, each data point performs the same amount of work (i.e., $w = w_i$), so we have $W = n \times w$. We note that w can be measured with different metrics: compute-intensive applications can use the number of floating-point operations; memory-bound applications can use the number of memory accesses; applications which are iterative in nature (e.g., scientific applications) can use the number of iterations. Let β be the fraction of data points assigned to the GPU (β shows the partitioning point). The workload size for the GPU ($W_G = n \times \beta \times w$) and the CPU ($W_C = n \times (1 - \beta) \times w$) can be expressed by β. Next, we use P to quantify the processing throughput of a processor for a given workload. P, defined as amount of workload processed per second, evaluates the processor's hardware capability. Therefore, we have $T_G = W_G/P_G$ and $T_C = W_C/P_C$.

Further, we use data-transfer size (O) and data-transfer bandwidth (Q, measured in bytes per second) to quantify the data-transfer time: $T_D = O/Q$[1]. There are three possible types of data-transfer scenarios: no data transfer (ND), partial data transfer (PD), and full data transfer (FD). ND is the ideal case in which there is negligible/no data transfer to the GPU. PD means that O is proportional to the partitioning point β, and therefore a larger GPU partition requires a larger data transfer. FD is the case that O is independent of β, thus no matter where the partitioning point is, the data-transfer size is fixed. The data-transfer type indicates whether to add T_D into the optimality equation, and shows the relationship between O and β.

After substituting the modeling of T_G (W_G/P_G), T_C (W_C/P_C), and T_D (O/Q) into the optimality equation, and performing equation transformations, we have:

$$\frac{W_G}{W_C} = \frac{P_G}{P_C} \times \frac{1}{1 + (O/W_G) \times (P_G/Q)} \tag{5.1}$$

Equation 5.1 is the partitioning model that indicates the optimal partitioning. If the data-transfer type is ND ($O = 0$), Equation 5.1 reduces to $W_G/W_C = P_G/P_C$. As W_G, W_C, and O are expressed by β, β is thus determined by the ratio of GPU throughput to CPU throughput (P_G/P_C, the relative hardware capability) and the ratio of GPU throughput to data-transfer bandwidth (P_G/Q, the GPU computation to data transfer gap). Knowing the two ratios, we can determine β from Equation 5.1.

[1] A full equation should be $T_D = O/Q + latency$, where latency is determined by the physical connection between the CPU and the GPU. Our measurement shows that, for PCIe connection, the latency (smaller than 100 μs) has negligible impact on T_D, thus can be ignored.

5.2.2 Predicting the Optimal Partitioning

To predict β, we need to estimate P_G/P_C and P_G/Q. The case studies in Section 5.1 already show that the two ratios are application dependent, thus we use profiling (i.e., we run the given application and dataset on the CPU and the GPU separately, and record their execution time profiles) to make a realistic estimation. We note that profiling is able to respond quickly and efficiently to application, dataset, and platform changes, because it models these aspects together.

Partial vs. Full Profiling

Observing that profiling for application kernels with a large number of identical iterations is usually time-consuming, we use partial profiling to save profiling cost. Thus, we limit the number of iterations to v ($v_{min} \leq v \leq w$), and use this partial execution to estimate the application global behavior. The choice of v is a trade-off between profiling accuracy and cost. When lowering v below a minimum threshold (platform dependent), the estimation quality will not be acceptable.

Assuming the profiled workload is W^V ($W^V = n \times v$, $v = w$ in full profiling), and the profiled data-transfer size is O^V (the full data-transfer size as we run the whole dataset on the GPU), we can estimate P_G/P_C and P_G/Q as:

$$\frac{P_G}{P_C} \approx \frac{W^V/T_G^V}{W^V/T_C^V} = R_{GC}, \quad \frac{P_G}{Q} \approx \frac{W^V/T_G^V}{O^V/T_D^V} = \frac{W^V}{O^V} \times R_{GD} \tag{5.2}$$

Substituting Equation 5.2 into Equation 5.1, we get three β solvers[2] corresponding to three data-transfer types:

$$\beta = \frac{R_{GC}}{1 + R_{GC}} \quad (0 < \beta < 1) \quad \text{if } O = O^V = 0 \tag{5.3a}$$

$$\beta = \frac{R_{GC}}{1 + R_{GC} + \frac{v}{w} \times R_{GD}} \quad (0 < \beta < 1) \quad \text{if } O = O^V \times \beta \tag{5.3b}$$

$$\beta = \frac{R_{GC} - \frac{v}{w} \times R_{GD}}{1 + R_{GC}} \quad (\beta < 1) \quad \text{if } O = O^V \neq 0 \tag{5.3c}$$

By measuring the time ratios T_C^V/T_G^V (noted as R_{GC}) and T_D^V/T_G^V (noted as R_{GD}), we can predict β. We note that it is necessary to know the ratio of v to w only when we enable partial profiling; if full profiling is used, this ratio will be 1, and there is no need to examine the code and get the values of v and w. Comparing Equations 5.3b and 5.3c to 5.3a, we see that the term $\frac{v}{w} \times R_{GD}$ actually quantifies the impact of data transfer, which makes β smaller than that without data transfer.

[2] In this work, we use "β solver" to refer to the solution to β.

Online vs. Offline Profiling

We have so far focused on online profiling, where we take the given problem size as input, profile the application one time, and measure R_{GC} and R_{GD} directly. We point out that offline profiling is also an option.

For offline profiling, we first profile the application with multiple sample problem sizes n_j. The profiled T_{Gj}, T_{Cj}, and T_{Dj}, together with n_j, are collected as training data. Next, we use curve fitting on the training data to build linear regression models $T_G(n)$, $T_C(n)$, and $T_D(n)$, assuming a linear correlation between n and each T. We further assume that problem sizes falling into different memory hierarchies may have non-uniform performance behavior, so we divide the full range of problem size into several sub-ranges (called n-sets) according to the platform's memory hierarchy. We generate a regression model for each n-set. For the current mainstream CPUs and GPUs, we use the six n-sets depicted in Figure 5.3.

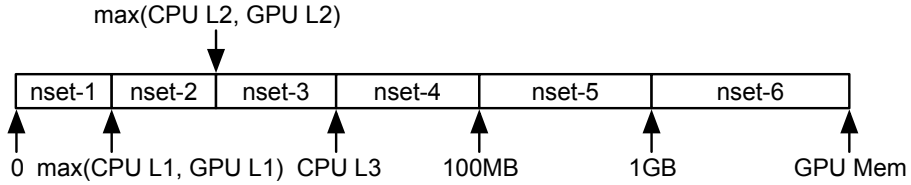


Figure 5.3: We divide the full range of problem size into six n-sets, from within the maximum of CPU L1 and GPU L1 cache sizes, to within the GPU main memory size.

The regression models can be summarized as follows:

$$T_G(n) \approx a_G^k \times n + b_G^k \quad n \in n\text{-set } k \tag{5.4a}$$
$$T_C(n) \approx a_C^k \times n + b_C^k \quad n \in n\text{-set } k \tag{5.4b}$$
$$T_D(n) \approx a_D^k \times n + b_D^k \quad n \in n\text{-set } k \tag{5.4c}$$

Linear correlation holds for all the n-sets, but the coefficients a_G, b_G, a_C, b_C, a_D, and b_D are different for different n-sets. Finally, we model $R_{GC}(n)$ and $R_{GD}(n)$ for each n-set k using the obtained regression models:

$$R_{GC}(n) = \frac{a_C^k \times n + b_C^k}{a_G^k \times n + b_G^k}, \quad R_{GD}(n) = \frac{a_D^k \times n + b_D^k}{a_G^k \times n + b_G^k} \tag{5.5}$$

For a new problem size, we check to which n-set it belongs, choose the corresponding models to predict its R_{GC} and R_{GD} (Equation 5.5), and then calculate its β (Equations 5.3a-5.3c).

We set the minimum sampling interval (i.e., $n_{j+1} - n_j$) at warp/wavefront size, as this is the smallest thread scheduling unit on GPUs and it is likely to facilitate mem-

ory accesses aligned to cache lines. Coarsening the sampling interval to a multiple of warp/wavefront size is acceptable, and saves profiling time, but *may* cause less accurate modeling. In Section 5.3.2, we further study the impact of coarsening on different n-sets.

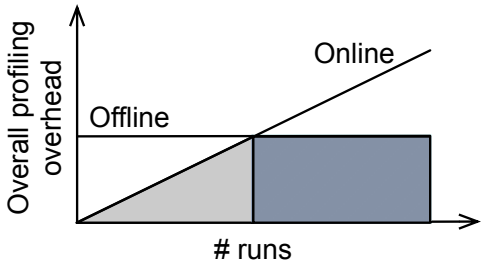


Figure 5.4: For offline profiling and online profiling, the relationship between the overall profiling overhead and the number of application runs with different problem sizes. In the light-shaded triangle area, online profiling has smaller overhead than offline profiling; in the dark-shaded rectangle area, offline profiling has smaller overhead.

Figure 5.4 illustrates the comparison between offline profiling and online profiling in terms of overall profiling overhead. For offline profiling, the overhead is constant because the number of samples is fixed. For online profiling, the overhead grows with the number of application runs. In general, if the execution scenario is one application repeatedly used with a wide variety of problem sizes (cases in the dark-shaded rectangle area), it is better to choose offline profiling. If one application has only a few interesting problem sizes (cases in the light-shaded triangle area), online profiling is more suitable. In our approach, we enable both profiling options to provide the users with the flexibility for choosing the one that fits their use cases.

5.2.3 Making the Decision in Practice

After obtaining β, we need to convert it into an actual hardware configuration. We first calculate the GPU partition size ($N_G = n \times \beta$) and CPU partition size ($N_C = n \times (1-\beta)$) using β. To prevent unnecessary waste of hardware computing power, we set the lower bound of N_G and N_C to half of the total GPU cores, and the total CPU cores, respectively. This is a non-aggressive estimation of the usage of hardware computing power. Next, we compare N_G and N_C to their lower bounds. If N_G is smaller than its lower bound, the GPU is considered to be underutilized, thus the decision is to use Only-CPU; for the same reason, if N_C is smaller than its lower bound, we will choose Only-GPU; if none of these condition is met, the decision is to use CPU+GPU. We will further round up N_G to a multiple of warp/wavefront size to improve the utilization of GPU cores, as recommended by [5, 90]. Accordingly, the final N_C is calculated ($n - N_G$).

We also note that for the FD data-transfer type, the predicted β can be smaller than zero, case in which the data-transfer impact is so large that it cancels out any GPU speedup. In this case, we directly decide to use the CPU.

Moreover, we adopt an incremental learning method based on historical data to avoid unnecessary online profiling, prediction, and decision-making. Assuming the application and hardware platform are given, for every tested problem size (n), its corresponding partitioning result and hardware configuration are stored in a repository. This information is used to learn the n-range for each type of hardware configuration. Once a new problem size is input, we check if it falls into the range of the Only-CPU or Only-GPU configuration, and, if so, we directly use the prior decision without going through the online profiling, prediction, and decision-making process. Only when the new input falls into the range of the CPU+GPU configuration, the whole process is triggered and completed, and the new partitioning and hardware configuration are added into the repository, further refining the n-range.

5.2.4 Extension to CPU+Multi-GPUs

In this subsection, we extend our approach to CPU+Multi-GPU platforms. We consider two different settings, in which the GPUs are identical or non-identical.

Identical GPUs

We find that our approach can be immediately extended to a platform with multiple *identical* GPUs. Assuming there are m GPUs connected to the host CPU, and their ratios (noted as R_{GC}^{One} and R_{GD}^{One}) are identical to each other, we first regard all the GPUs as an aggregated GPU, and we have:

$$\begin{cases} R_{GC} = m \times R_{GC}^{One} & \\ R_{GD} = R_{GD}^{One} & \text{if } O = O^V \times \beta \\ R_{GD} = m \times R_{GD}^{One} & \text{if } O = O^V \neq 0 \end{cases} \tag{5.6}$$

The aggregated GPU's R_{GC} is increased by m times as the GPU computing power is aggregated. For R_{GD}, depending on the data-transfer type, the calculation varies. For the PD type, R_{GD} is the same as that of each GPU because the data-transfer work is equally shared by all the GPUs. For the FD type, R_{GD} is increased by m times as the same amount of data has to be copied to all the GPUs. By substituting Equation 5.6 into Equations 5.3a-5.3c, we calculate the aggregated partitioning point (β_m), and each GPU gets $1/m \times \beta_m \times n$ work. Thus, by measuring the ratios of one GPU, we are able to get the partitioning for multiple GPUs.

We note that this solution is especially useful when a single GPU cannot hold all the data assigned to it due to the memory capacity limitation. We can make the CPU perform more work, which leads to potential performance loss. Alternatively, we can use multiple GPUs without sacrificing performance. By using partial profiling (because of the memory limitation, we cannot use full profiling), we can obtain the ratios for one GPU, and eventually derive the right decomposition for the multi-GPU configuration.

Non-identical GPUs

For the case of *non-identical* GPUs, we extend the definition of the optimal partitioning to cope with multiple processing units, i.e., $T_C = T_{G1} + T_{D1} = T_{G2} + T_{D2} = \cdots$. To simplify the presentation, we assume a platform consists of one CPU and two different GPUs. Let β_C, β_{G1}, and β_{G2} denote the fractions of data points assigned to the CPU, GPU_1, and GPU_2, respectively. The optimality equation is expressed as:

$$\beta_C + \beta_{G1} + \beta_{G2} = 1 \tag{5.7a}$$

$$\frac{W_G}{P_C} = \frac{W_{G1}}{P_{G1}} + \frac{O_{G1}}{Q_{G1}} = \frac{W_{G2}}{P_{G2}} + \frac{O_{G2}}{Q_{G2}} \tag{5.7b}$$

Similarly, we use profiling to estimate P_C, P_{G1}, Q_{G1}, P_{G2}, and Q_{G2}. Substituting the estimations into Equation 5.7b, we have:

$$\text{if } O_{G1} = O_{G2} = O^V = 0\text{:}$$
$$\frac{w}{v}T_C^V\beta_C = \frac{w}{v}T_{G1}^V\beta_{G1} = \frac{w}{v}T_{G2}^V\beta_{G2} \tag{5.8a}$$
$$\text{if } O_{G1} = O^V\beta_{G1}, O_{G2} = O^V\beta_{G2}\text{:}$$
$$\frac{w}{v}T_C^V\beta_C = \frac{w}{v}T_{G1}^V\beta_{G1} + T_{D1}^V\beta_{G1} = \frac{w}{v}T_{G2}^V\beta_{G2} + T_{D2}^V\beta_{G2} \tag{5.8b}$$
$$\text{if } O_{G1} = O_{G2} = O^V \neq 0\text{:}$$
$$\frac{w}{v}T_C^V\beta_C = \frac{w}{v}T_{G1}^V\beta_{G1} + T_{D1}^V = \frac{w}{v}T_{G2}^V\beta_{G2} + T_{D2}^V \tag{5.8c}$$

Except β_C, β_{G1}, and β_{G2}, the other terms are known. Thus, we can use one β to express the other two (e.g., we use β_C to express β_{G1} and β_{G2}), and finally solve Equation 5.7a.

In fact, our approach can be extended to a different mix of hardware accelerators, connected to the host with either a shared memory or a physical connection (e.g., CPUs, GPUs, FPGAs, Xeon Phi), as we only model the accelerator by its capability to process the workload, without using any architecture details.

Summary: Our systematic approach uses modeling and prediction techniques to determine the optimal partitioning for a given application, dataset, and platform, and adjusts

it to the right hardware configuration with practical consideration. The way to use our approach is simple: the users just need to model the application workload (for balanced applications with full profiling, only the problem size n is needed), profile the platform (obtaining R_{GC} and R_{GD}), use the solver to determine β (in O(1) time), and finally choose the hardware configuration.

Implementation-wise, the users need to implement the application for both CPUs and GPUs, and enable the profiling and the partitioning (i.e., workload distribution and output gathering) in the code. The resulting code can be reused with little effort for different datasets and platforms. Thus, by using our approach, one obtains a reusable single-device and multi-device code of the application, a hardware configuration per dataset and platform, improved application performance, and more efficient hardware utilization.

5.3 Experimental Evaluation

In this section, we present a detailed evaluation of our systematic approach.

5.3.1 Experimental Setup

The evaluation focuses on five important aspects: (1) *The validation* (Section 5.3.2) examines the quality of our approach, checking if it predicts the optimal partitioning and selects the right hardware configuration; (2) *The performance analysis* (Section 5.3.3) studies how much performance is gained through a partitioned execution compared to a single-processor execution; (3) *The impact of data transfer* (Section 5.3.4) quantifies how the data transfer affects the partitioning and the performance; (4) *The adaptiveness* (Section 5.3.5) checks if our approach has the ability to adapt to application, dataset, hardware, and software changes; (5) *The applicability on CPU+Multi-GPUs* (Section 5.3.6) proves how our approach is applied to CPU+Multi-GPU platforms. According to the five main aspects, we select a set of applications, datasets, hardware and software platforms, which are further explained below.

Applications and Datasets

We have used 12 balanced applications listed in Table 5.2. DotProduct (SDOT), Matrix-Vector Multiplication (SGEMV), and Matrix-Matrix Multiplication (SGEMM) are linear algebra applications typical in libraries like BLAS. Black-Scholes (BS), Mersenne Twister (MT), and Inverse Cumulative Normal Distribution (ICND) are finance-related algorithms for evaluation of European option prices, for generation of pseudorandom numbers, and for generation of inverse cumulative normal distributions, respectively. Separable Convolution (SConv), DCT8x8, and Sobel Filter (SF) are image processing algorithms

useful for image sharpening, blurring, compression, and edge-detection, etc. FDTD3d is a finite difference time domain solver which applies 3D stencil computation. NBody and Particle Collision (PC) are scientific simulation of interactions of individual bodies, in the form of all-pair gravitational attraction and local elastic collision, respectively. The 12 applications are widely used in different domains, and have different workload characteristics. For example, SGEMM, BS, ICND, NBody, and PC are compute intensive as these applications perform a large number of floating-point operations, while the others tend to be memory bound.

Table 5.2: The application, its use domain, and type.

Application	Domain	Type
SDOT	Linear algebra	Memory-bound
SGEMV	Linear algebra	Memory-bound
DCT8x8	Image processing	Memory-bound
BS	Finance	Compute-intensive
MT	Finance	Memory-bound
FDTD3d	Numerical methods	Memory-bound
SConv	Image processing	Memory-bound
ICND	Finance	Compute-intensive
PC	Scientific simulation	Compute-intensive
SF	Image Processing	Memory-bound
SGEMM	Linear algebra	Compute-intensive
NBody	Scientific simulation	Compute-intensive

For each balanced application, we use six different problem sizes ($n1$–$n6$), with each falling into one of the six n-sets (see Figure 5.3), to study how online profiling responds to the change of problem sizes. This is done by varying the vector and/or array size in SDOT, SGEMV, and SGEMM; the option count in BS; the number of random numbers in MT and ICND; the image size in SConv, DCT8x8, and SF; the number of bodies in NBody and PC; and the grid size in FDTD3d. Therefore, we have a total of 72 test cases. To validate offline profiling, multiple problem sizes with a minimum sampling interval of 32 data points are used.

The implementations of balanced applications are provided by Nvidia OpenCL SDK. Thus, CPUs and GPUs share the same code structure and parallelization approach. We further make the following modifications per application: for the kernel code (on the device side), we apply architecture-friendly optimizations (e.g., using row-major memory accesses on CPUs and column-major memory accesses on GPUs) [112] to improve application performance on each processor; for the host code, we enable the profiling and the partitioning to use all the processors when necessary.

Hardware and Software Platforms

We mainly perform our evaluation on a hardware platform S4, which integrates an Intel Xeon E5-2620 CPU (hyper-threading enabled) and an Nvidia Tesla K20 GPU. Table 5.3 lists the hardware information. To evaluate the adaptiveness of our approach to hardware changes, we further use three other platforms, S1, S2, and S3 (see Table 4.2): S1 has a dual-socket Intel E5645 six-core CPU and an Nvidia Tesla C2050 GPU; S2 has the same CPU as S1 and an Nvidia Quadro 600 GPU; S3 has the same GPU as S1 and a dual-socket Intel E5620 quad-core CPU. To apply our approach on CPU+multi-GPU platforms, we use platforms S5 and S6: S5 has a dual-socket Intel X5650 six-core CPU and two identical GPUs (Nvidia GTX 580); S6 has a dual-socket Intel E5645 six-core CPU and two non-identical GPUs (Nvidia Tesla C2050 and Quadro 600).

Table 5.3: The hardware components of the S4 platform.

	CPU	GPU
Processor	Intel Xeon E5-2620	Nvidia K20m
Frequency (GHz)	2.0	0.705
#Cores	6 (12 as HT enabled)	2496 (13 SMXs)
Peak GFLOPS (SP/DP)	384.0/192.0	3519.3/1173.1
L1 cache size (KB)	32×6	16×13
L2 cache size (KB)	256×6	1536
L3 cache size (MB)	15	No L3 cache
Memory capacity (GB)	64	5
Peak Memory Bandwidth (GB/s)	42.6	208.0

The device-side compilers are Intel OpenCL SDK 2013 and NVIDIA OpenCL SDK 5.5, respectively. The host-side compiler is GCC 4.4.6 with -O3 option. We also update the device-side compilers to test how our approach reacts to software changes.

5.3.2 Validation

First, we validate if our approach predicts the optimal partitioning and selects the right hardware configuration. We follow the steps specified in Sections 5.2.2 and 5.2.3: estimate R_{GC} and R_{GD} by profiling, calculate the partitioning point β through the solver, and decide the final hardware configuration taking the lower bounds of the partition sizes into account. Table 5.4 summarizes the experimental results.

We note that two data transfer types, PD and FD, coexist in SGEMV ($y = A \times x$) and SGEMM ($C = A \times B$). Vector x and Matrix B are copied to both processors (FD), while all the other memory buffers (the majority) are split (PD). In this case, we choose PD as the main data transfer type to select the solver.

Table 5.4: The profiling results (R_{GC} and R_{GD}), the partitioning point (β), and the final hardware configuration decision (highlighted with different gray scales) per application and problem size. OC, OG, and CG represent the configuration of Only-CPU, Only-GPU, and CPU+GPU, respectively.

	Application	SDOT	SGEMV	DCT8x8	BS	FDTD3d	MT	SConv	ICND	PC	SF	SGEMM	Nbody
$n1$	R_{GC}	7.91	3.08	6.67	4.67	5.71	4.16	1.83	6.00	1.44	5.29	6.04	1.64
	R_{GD}	52.36	45.25	47.42	27.25	16.69	12.88	9.00	12.36	1.17	1.69	26.54	1.31
	β	0.1291	0.0625	0.1210	0.1418	0.2439	0.2324	0.1543	0.3099	0.3986	0.6631	0.1799	0.4149
	Decision	OC	OC	OC	OC	OC	OC	OC	CG*	OC	CG	OC	OC
$n2$	R_{GC}	2.27	5.35	5.07	8.64	5.16	3.65	3.25	14.35	1.87	6.57	4.78	5.17
	R_{GD}	57.07	64.55	47.13	63.93	25.76	15.51	20.65	53.18	0.98	2.00	5.07	0.15
	β	0.0376	0.0755	0.0952	0.1175	0.1616	0.1816	0.1307	0.2094	0.4851	0.6866	0.4406	0.8180
	Decision	OC	OC	OC	CG*	OC	OC	OC	CG	CG	CG	CG	CG
$n3$	R_{GC}	1.53	2.43	11.27	12.01	7.88	2.96	4.22	23.40	2.40	6.52	4.15	4.55
	R_{GD}	57.51	46.75	81.62	85.10	30.65	8.38	13.87	59.86	1.30	2.03	0.70	0.01
	β	0.0255	0.0484	0.1201	0.1225	0.1993	0.2402	0.2212	0.2777	0.5100	0.6825	0.7090	0.8183
	Decision	CG*	OC	OC	CG	OC	OC	OC	CG	CG	CG	CG	CG
$n4$	R_{GC}	2.98	3.60	13.36	13.97	9.02	2.99	4.91	24.21	1.46	6.79	4.88	4.71
	R_{GD}	47.77	48.12	74.60	68.37	32.26	6.83	11.23	56.40	0.18	2.18	0.29	0.001
	β	0.0576	0.0683	0.1502	0.1676	0.2133	0.2734	0.2867	0.2967	0.5526	0.6809	0.7916	0.8248
	Decision	CG	OC	CG	CG	CG	CG	CG	CG	CG	CG	CG	CG
$n5$	R_{GC}	3.12	4.03	12.46	14.10	12.66	3.04	5.02	24.90	1.58	6.86	5.02	4.74
	R_{GD}	45.65	51.09	71.25	62.93	31.03	6.71	10.53	57.04	0.38	2.27	0.09	0.0001
	β	0.0626	0.0719	0.1471	0.1807	0.2833	0.2813	0.3031	0.3002	0.5340	0.6769	0.8220	0.8258
	Decision	CG	CG	CG	CG	CG	CG	CG	CG	CG	CG	CG	CG
$n6$	R_{GC}	3.22	4.01	12.32	14.14	13.11	3.00	5.05	25.07	1.83	6.95	4.99	4.75
	R_{GD}	45.98	51.22	69.61	62.62	29.96	6.69	10.44	57.09	0.30	2.29	0.04	0.00005
	β	0.0641	0.0712	0.1486	0.1818	0.2975	0.2813	0.3064	0.3015	0.5837	0.6790	0.8271	0.8261
	Decision	CG	CG	CG	CG	CG	CG	CG	CG	CG	CG	CG	CG

* The right configuration is Only-CPU. Ours is CPU+GPU because we set a non-aggressive lower bound for N_G.

We see that R_{GC} and R_{GD} vary for different applications and problem sizes, proving that it is necessary to use profiling to estimate the relative hardware capability and the GPU computation to data transfer gap, as profiling is sensitive to application and problem size changes.

To validate our approach, we compare the predicted hardware configuration to an oracle hardware configuration obtained by auto-tuning. The auto-tuning (based on a binary search) starts from $\beta = 0.5$, tunes β until the condition $T_G + T_D = T_C$ is met, and returns the configuration of CPU+GPU with the partition sizes (N_G and N_C). In practice, there is a threshold δ applied to control the auto-tuning process (i.e., when $|T_G + T_D - T_C| \leq \delta$, the auto-tuning is stopped, and the resulting partitioning is considered as the optimal one). Once N_G or N_C is smaller than its lower bound, the auto-tuning returns the Only-CPU or Only-GPU configuration.

We find that our prediction and the auto-tuning determine practically the same β (thus, the same hardware configuration) in 62 out of 72 test cases. Figure 5.5 shows the performance comparison. In these 62 cases, 22 cases adopt the Only-CPU configuration, therefore there is no performance difference between our prediction and the auto-tuning; 40 cases adopt the CPU+GPU configuration, and the relative performance difference is within 10%. For the other 10 cases, the predicted β is a bit off compared to the auto-tuned β. However, the prediction and the auto-tuning still reach the same decision, CPU+GPU, in 7 cases (SDOT-$n4, 5$, DCT8x8-$n4$, BS-$n3, 4$, ICND-$n3$, and SF-$n1$), where the relative performance difference is within 37% (the absolute performance difference is within 2 ms). The remaining 3 cases are SDOT-$n3$, BS-$n2$, and ICND-$n1$, marked with (*) in Table 5.4, in which the auto-tuning decision is Only-CPU. The prediction selects CPU+GPU because we set a non-aggressive lower bound for N_G. Actually, the performance of using the mix is very close to that of using only the CPU: within 0.3 ms. We note that in most test cases, the performance difference between the auto-tuning and the prediction is within the threshold δ we set, i.e., the prediction is *as good as* the optimal partitioning.

The Validation of Partial Profiling

To validate the use of partial profiling, we use SGEMM-$n5, 6$, and halve the width of Matrix A and the height of Matrix B (thus, $\frac{v}{w} = \frac{1}{2}$, while the number of data points need computing in Matrix C remains the same). The partitioning point we get is very close to that with full profiling and that with auto tuning, leading to at most 4% difference in performance and 50% time saved in profiling. If we decrease the partial profiling ratio to $\frac{v}{w} = \frac{1}{8}$, we can save more than 80% time in profiling, but the performance difference is increased to 11%, showing the trade-off between profiling accuracy and cost.

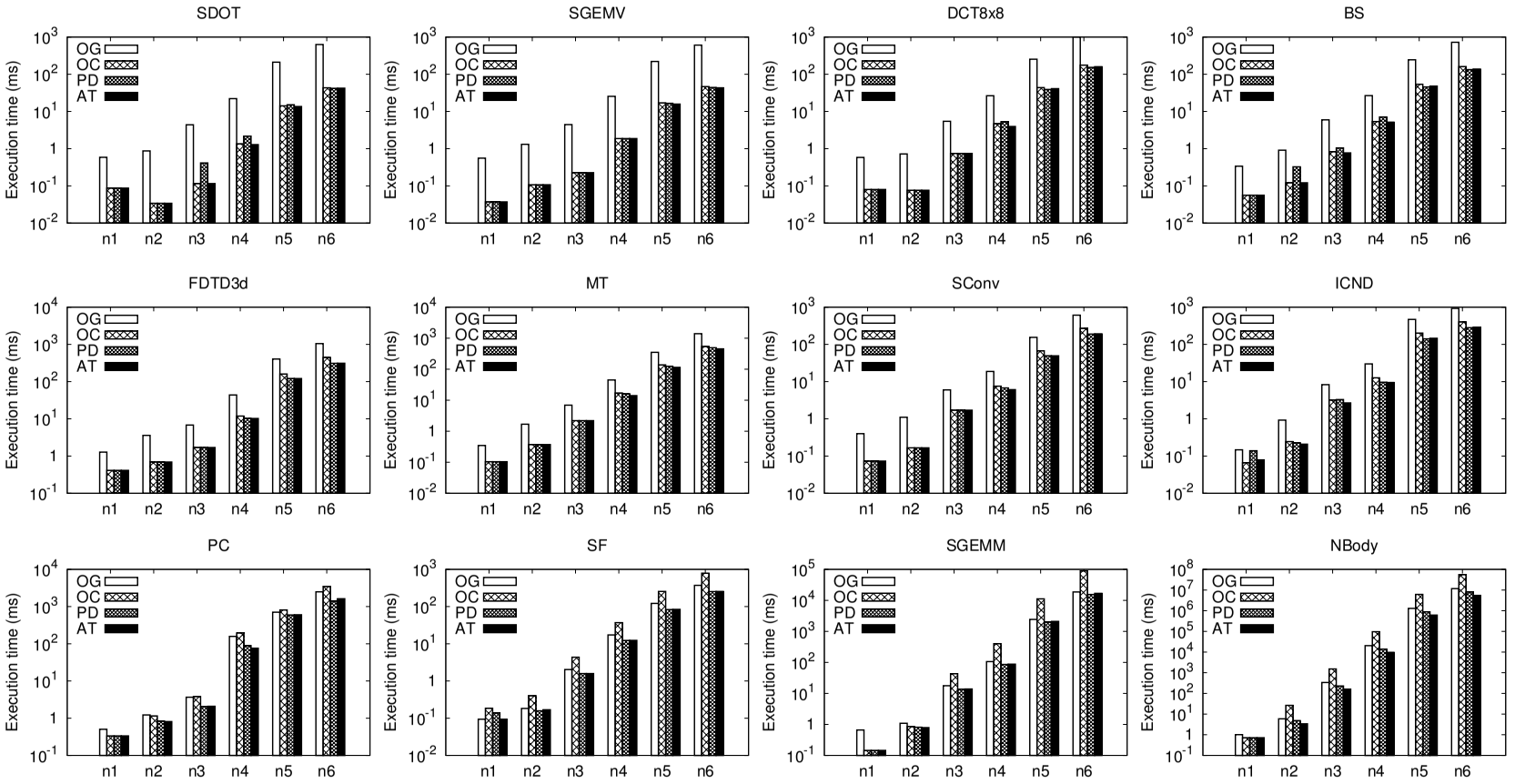


Figure 5.5: The execution time of the predicted (PD), auto-tuned (AT), Only-GPU (OG), and Only-CPU (OC) hardware configurations per application. The horizontal axis is the problem size. The vertical axis is the execution time (in ms). The 12 figures have different log scales. In the 3 cases (SDOT-$n3$, BS-$n2$, and ICND-$n1$), it seems that PD has "much larger" execution time than AT and OC, but the actual difference is within 0.3 ms.

The Validation of Offline Profiling

To validate the use of offline profiling, we use BS and SGEMM, as they are representative applications in which the CPU and the GPU take most of the computation, respectively. We set the minimum sampling interval to 32 (the warp size of Nvidia GPUs), and coarsen the sampling interval to a multiple of 32 until each n-set has a maximum of 40 problem sizes. We note that 40 samples per n-set (a total of 240 samples) are sufficient for curve fitting in our empirical experiment. We profile all the samples in each n-set, and record their execution time profiles.

Before performing the curve fitting, we first directly compute each sample's R_{GC} and R_{GD}. Figure 5.6 shows the variation. We see that R_{GD} has larger variation than R_{GC}. More importantly, larger n-sets (n-set5,6 in BS and n-set4,5,6 in SGEMM) have smaller or even invisible variation compared to smaller n-sets. R_{GC} and R_{GD} also stabilize across larger n-sets. Therefore, we can use only a few/even one sample(s), and use the average/the one R_{GC} and R_{GD} as an estimation for larger n-sets. In this case, we only need to perform the curve fitting for smaller n-sets, saving the profiling and training time.

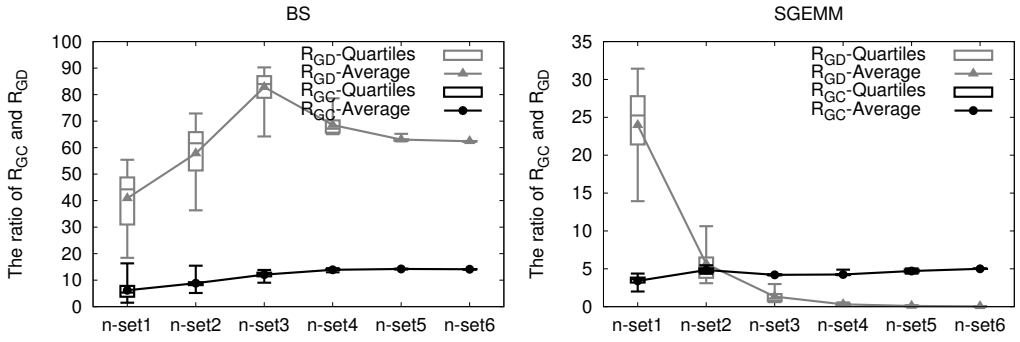


Figure 5.6: Variation of R_{GC} and R_{GD}, in box-and-whisker plot, per n-set.

Next, we fit linear regression models $T_G(n)$, $T_C(n)$, and $T_D(n)$, and build the models $R_{GC}(n)$ and $R_{GD}(n)$ for each n-set as we described in Section 5.2.2. We randomly generate 10 test problem sizes falling into different n-sets. For each test problem size, we choose its corresponding models to predict its R_{GC} and R_{GD}, and finally predict the partitioning point β. The average performance difference between the predicted partitioning and the auto-tuned partitioning is 15% and 5% for BS and SGEMM, respectively. For problem sizes within larger n-sets (n-set5,6 in BS and n-set4,5,6 in SGEMM), we also disable curve fitting, and use the average R_{GC} and R_{GD} of the samples as an estimation for prediction. The predicted β is very close to that obtained using curve fitting, leading to minimal performance difference.

5.3.3 Performance Analysis

We compare the performance of the hardware configuration obtained through our prediction (with online profiling) to that of the Only-GPU and Only-CPU configurations (see Figure 5.5).

Compared to Only-GPU, the average performance gain of using our configuration for SDOT and SGEMV is 13.7× and 14.6×, respectively. The performance improvement is significant because the data transfer severely diminishes the performance of Only-GPU. Our approach detects that R_{GD} is very large, and assigns more than 90% of the work to the CPU. For DCT8x8 and BS, R_{GD} is also very large, but R_{GC} is increased by 2×–4× compared to that of SDOT and SGEMV, thus the performance improvement is 7.0× and 4.8×, respectively, as more work is shifted to the GPU. For FDTD3d, MT, and SConv, their R_{GC} and R_{GD} are moderate, leading to a moderate performance improvement of 3.2×–4.1×. On the contrary, ICND has both large R_{GC} and R_{GD}, and its performance improvement is around 2.9×. PC, SF, SGEMM, and NBody show the smallest performance improvement (1.2×–1.8×). As R_{GD} is very small in these applications, more than 50% (up to 83%) of the work is actually assigned to the GPU. We further notice that the hardware configuration in all the test cases is either Only-CPU or CPU+GPU (see Table 5.4). Thus, if we would only use the GPU for computation, performance penalties of up to 96% would be observed for applications with large R_{GD} and small R_{GC} (e.g., SDOT and SGEMV).

Compared to Only-CPU, our configuration gets small performance speedup for SDOT, SGEMV, DCT8x8, BS, FDTD3d, MT, SConv, and ICND (less than 1.4×). This is because the CPU takes most of the work after the partitioning. For SDOT-$n3$, BS-$n2$, and ICND-$n1$, the cases marked with (*) in Table 5.4, it seems that Only-CPU largely outperforms ours, but the actual difference is within 0.3 ms (see Section 5.3.2). For PC, SF, SGEMM, and NBody, the performance speedup is 2.0×–6.6×. Our configuration largely improves the performance as ours correctly assigns most of the work to the GPU.

5.3.4 The Impact of Data transfer

We quantify the data transfer impact by comparing the partitioning with and without data transfer. We assume that there is no data transfer between the GPU and the CPU, and we use Equation 5.3a to predict the new β without data transfer.

We see that the new β is shifted towards 1, making most test cases choose the CPU+GPU configuration. The impact of data transfer varies. For SDOT, SGEMV, DCT8x8, BS, and ICND (the ones with higher R_{GD}), up to 80% of the work is moved from the CPU to the GPU, resulting in up to 87% change in performance (the performance is increased because the data-transfer time is not taken into account). For FDTD3d, MT, and SConv (the ones with moderate R_{GD}), the GPU gets on average 58% more work, and

the performance change is around 50%. For PC, SF, SGEMM, and NBody, the change of β is small (on average 14%) . As the original data-transfer overhead is already very small, the performance change is around 21%.

Applications with higher R_{GD} (a larger gap between GPU computation throughput and data-transfer bandwidth) suffer more performance impact from data transfer. In this situation, an integrated CPU+GPU platform (like AMD APU [4, 21] and Intel Sandybridge [51]) can be a good hardware solution, as it largely eliminates the data-transfer overhead. A software solution is to attain high data-transfer bandwidth using pinned memory [127]. For example, we have actually used pinned memory in SF to decrease its R_{GD}. As a result, the majority of the work is assigned to the GPU, and the overall performance is increased. Another software solution is to hide data transfer by overlapping it with computation [79], but the implementation may not be easy for every application.

5.3.5 Adaptiveness

The above experiments have shown that our method is able to respond to application and problem size changes. In this subsection, we evaluate its adaptiveness to hardware and software (i.e., compiler) changes by using SGEMM.

The adaptiveness to hardware changes: We use the S1 platform (see Table 4.2) as the reference platform, and perform the profiling and the partitioning. Then, we use the S2 platform, which has the same CPU as S1 but a less powerful GPU (i.e., less cores, lower peak GFLOPS, and smaller memory capacity). Our approach responds to the GPU degradation by re-profiling the GPU and recalculating R_{GC} and R_{GD}. As a result, the new β is shifted towards the CPU, making the CPU process more work. Next, we use the S3 platform, where the GPU is the same and the CPU is changed to a lower configuration compared to S1. Similarly, our approach reacts to the CPU change by re-profiling the CPU and re-assigning more work to the GPU accordingly.

The adaptiveness to software changes: As Nvidia and Intel release new versions of their compilers frequently, we want to evaluate the impact of the compiler changes. We use three versions of the Intel compiler (v2012, v2013, and v2014) and three versions of the Nvidia compiler (v4.2, v5.5, and v6.5). We find that the Intel compiler improves application performance (on the CPU) from v2012 to v2013, while the Nvidia compiler (the OpenCL part) and the Intel compiler v2014 do not. After changing the Intel compiler from v2012 to v2013, our approach re-profiles the application and re-predicts the partitioning point. The new partitioning makes the CPU get more work, leading to up to 13% performance improvement.

In summary, thanks to the use of profiling, our approach is able to adapt to new hardware, different applications, different problem sizes, and even compiler changes.

5.3.6 Applicability on CPU+Multi-GPUs

To test our approach on CPU+Multi-GPU platforms, we use SGEMM with the largest problem size ($n6$). We consider two situations, one CPU+two identical GPUs (the S5 platform), and one CPU+two non-identical GPUs (the S6 platform).

On S5, as the two GPUs are the same, our approach only needs to profile one GPU and derive the partitioning for two GPUs. By examining the execution time of each processor after partitioning, we validate the correctness of our approach. The performance of different hardware configurations is shown in Figure 5.7 (a). The configurations of CPU+GPU and CPU+2×GPUs are obtained using our partitioning approach. The configuration of 2×GPUs evenly divides the computation workload on each GPU. We see that CPU+2×GPUs improves the performance by 2.2×, 1.8× and 1.1× over 1×GPU, CPU+GPU, and 2×GPUs, respectively.

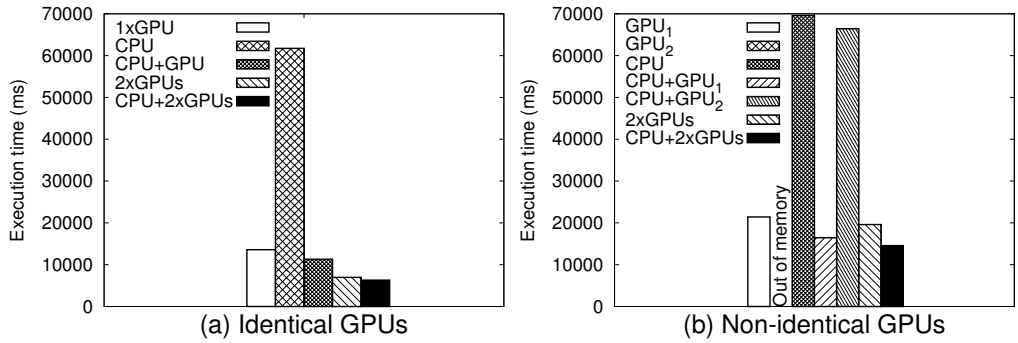


Figure 5.7: The execution time of different hardware configurations on (a) the S5 platform with identical GPUs, and (b) the S6 platform with non-identical GPUs. The vertical axis is the execution time (in ms).

On S6, we profile each processor to predict the partitioning. As the two GPUs are not identical, their partition sizes are different. GPU_1 with higher processing capability gets a larger partition (β_{G1} = 0.68), and GPU_2 gets a smaller partition (β_{G1} = 0.11). The performance of CPU+2×GPUs is improved by 32% and 12% compared to that of GPU_1 and CPU+GPU_1, respectively. We note that SGEMM-$n6$ exceeds the memory capacity of GPU_2, and our approach uses partial profiling to estimate its processing capability. When using CPU+GPU_2 or 2×GPUs, the predicted partition still exceeds the memory of GPU_2, so we make the CPU or GPU_1 share more work until the partition of GPU_2 fits its memory. However, this also lowers the overall performance as presented in Figure 5.7 (b). Therefore, using CPU+2xGPUs is an alternative solution without sacrificing performance.

5.4 Discussion

In this section, we discuss the features, the advanced usage, and the limitations of our approach.

By introducing the two metrics, the relative hardware capability and the GPU computation to data transfer gap, our approach captures the heterogeneity of the hardware platform. The use of profiling makes it possible to estimate the two ratios in the application and dataset context, which adapt to any changes in applications, datasets, hardware, and software. Taking these two metrics into modeling, our approach predicts the optimal partitioning, leading to the best performance out of a given heterogeneous platform.

Compared to dynamic partitioning [14, 103], which divides the whole workload into multiple chunks and dynamically distributes the chunks to processors, our static partitioning wins in two aspects. Firstly, dynamic partitioning introduces scheduling overhead (including multiple scheduling decisions, kernel invocations, and data transfers), which does not appear in ours. Secondly, dynamic partitioning cannot always determine the optimal partitioning between the processors as its first speculation might be off. On the contrary, our approach directly determines the optimal partitioning, and therefore the performance of our partitioning will not be worse than that of dynamic partitioning.

We note that our approach can assist hardware selection. For example, if the CPU is fixed, and one GPU must be selected from a set of options, we can compare the predicted β in each combination. If $\beta_i > \beta_j$, it means that GPU$_i$ makes the CPU process less work than GPU$_j$, leading to lower overall execution time. Thus, the GPU which gets the maximum β is the best choice, performance-wise.

In this chapter, we focus on single-kernel data-parallel applications with independent workload per data point. We point out that our approach can also be applied to applications in which several (adjacent or random) data points are needed to calculate one result (e.g., stencils) by replicating some/all data on both the CPU and the GPU, and applications with multi-phases (kernels) and inter-phase synchronizations by using our approach phase-by-phase.

We further note that our approach is designed for single-node heterogeneous platforms. For clusters and HPC systems which have accelerator-equipped nodes, our approach can be used, in combination with existing inter-node scheduling, for intra-node workload partitioning.

5.5 Related Work

Multiple studies [33, 44, 122, 128, 133] have proved that using heterogeneous platforms improves application performance and hardware utilization. Our approach is an essential improvement to these studies, because it enables high efficiency in using heterogeneous

platforms by (1) optimal workload partitioning and (2) hardware configuration recommendation.

Several static partitioning approaches have been developed to solve the optimal partitioning. For example, Merge [76] and the work of Tomov et al. [125] rely on programmer's performance estimation to determine the partitioning. The requirement of manual work largely limits the applicability of these solutions. Grewe et al. [37] built a prediction model that relates static program features to the optimal partitioning by using machine learning. Kofler et al. [67] further added dynamic program features (e.g., problem-size related features) into the model and extended the model to platforms with multiple identical GPUs. Both approaches can be used for a wide range of applications with different problem sizes but cannot quickly adapt to platform changes as the model is trained on the given platform. A lot of training data needs to be collected and used to build a prediction model, which is costly and must be repeated for platform changes. Qilin [78] is an analytical-model based partitioning approach. It uses offline profiling and curve fitting to model the relationship between the application execution time and the partition size, and determines the optimal partitioning based on the modeling. Compared to Qilin, our approach provides both online and offline profiling options for the users. We further refine the offline profiling process to increase the model accuracy. Moreover, Qilin does not differentiate the data-transfer time from the kernel-computation time, which also lowers its partitioning quality and the overall performance.

To obtain the best performance on a given hardware platform, performance modeling techniques [46, 81, 121] have been developed. However, these techniques model the application and the platform capabilities separately, and therefore they are not adaptive to new platforms, different application types (e.g., memory-bound or compute-intensive), and different datasets: a lot of effort is spent on benchmarking hardware, modeling applications, matching models, and calibration. Compared to these solutions, our approach models the application performance together with the dataset and the platform by using profiling, thus it easily reacts to application, dataset, and hardware changes at the price of a small extra overhead for (partial) profiling.

As alternative to static partitioning, dynamic partitioning [14, 103] aims to achieve load balancing on heterogeneous platforms at runtime. Although it works for many applications, it may lead to high scheduling overhead and suboptimal partitioning as reported in [37, 67].

Finally, inter-kernel dynamic scheduling approaches [8, 12, 27] have been proposed for applications with multiple kernels. Our static partitioning works on a different level (intra-kernel partitioning), and can be combined with these dynamic scheduling approaches to improve application performance.

5.6 Summary

Deploying data parallel applications with the right hardware configuration (Only-CPU, Only-GPU, or CPU+GPU) and the optimal partitioning (for the CPU+GPU configuration) is essential for improving application performance on heterogeneous platforms. A wrong hardware choice or workload decomposition may lead to significant performance loss. In this chapter, we propose a systematic approach to determine the best-performing workload partitioning and hardware configuration. This approach effectively captures the heterogeneity of the hardware platform, quickly and correctly determines the optimal partitioning, and significantly improves the application performance and hardware utilization. It is applicable to different applications and hardware mixes, and is user-friendly, as it does not require deep understanding of the application and the platform. Therefore, our work provides an effective, efficient, and adaptive approach for users who are looking for high performance and efficiency on heterogeneous platforms.

So far, we have proposed workload partitioning solutions for single-kernel data parallel applications with balanced (Chatper 5) or imbalanced workloads (Chapter 4). We will study applications with more complex kernel structures in Chapter 6, aiming to maximize the applicability of our workload partitioning for heterogeneous platforms.

Chapter 6

Maximizing the Performance and Applicability for Workload Partitioning

In this chapter, we investigate workload partitioning for single- and multi-kernel applications, with the aim to improve performance for a large variety of applications on heterogeneous platforms. We propose a set of partitioning strategies combining the best features of static and dynamic partitioning, and propose an application-driven method to select the best strategy for a given application based on its application kernel structure.

To improve application performance on heterogeneous platforms, we need to partition the workload to utilize each processor on the platform. However, how to perform the partitioning to achieve the best performance is not trivial. To tackle this challenge, we follow an application-centric way to determine the most suitable partitioning strategy for a given application.

Our work focuses on data parallel applications, where there is massive parallelism to exploit. Each parallel section of code in the program is called a *kernel*, and a data parallel application can have *one* or *multiple* kernels executed in certain sequences. In data parallel applications, the workload is proportional to the number of data points to be computed in the kernel (i.e., the problem size). Partitioning the workload is therefore equivalent with splitting the data and assigning partitions to different processors.

There are multiple ways to perform the partitioning. At high level, they can be categorized into *static* and *dynamic* partitioning. Static partitioning determines a fixed partitioning before runtime, dividing the data into several partitions, typically as many as processors. Dynamic partitioning divides the data into multiple partitions at runtime (usually more than the number of processors), and schedules the partitions to processors based on certain scheduling policies.

Previous studies have developed different static and dynamic partitioning strategies suitable for different data parallel applications. The work in [37, 38, 67, 71, 78, 117, 120]

proposed static methods; the difficulty in these studies is to determine the optimal partitioning by building prior knowledge. Moreover, these methods are only designed for single-kernel applications. Dynamic partitioning strategies for both single-kernel [14, 103, 108] and multi-kernel applications [8, 12, 27, 39] have been proposed, where the scheduling policy, the data dependency, and the locality between multiple kernels are the main research points. These strategies have wider applicability, but often lead to suboptimal performance due to the scheduling policy and the runtime scheduling overhead. In other words, static partitioning trades applicability for performance, while dynamic partitioning trades performance for applicability.

To satisfy both requirements—the applicability and the performance—in one go, we aim to design an application analyzer that chooses the best performing strategy for different types of applications. To design the application analyzer, we must answer two important questions.

First, *what is an appropriate application classification?* An appropriate classification makes it possible to propose efficient partitioning strategies. We note that simply using the number of kernels to classify the applications is not sufficient. In this work, we use the application kernel structure, i.e., the number of kernels as well as the kernel execution flow, to classify the applications into five classes.

Second, *what are efficient partitioning strategies?* We answer this question in two steps. First, we propose five partitioning strategies starting from existing successful static (Chapter 5) and dynamic solutions [27]. Specifically, we propose new ways to apply static partitioning to multi-kernel applications, and only use dynamic partitioning as a fall-back scenario. Next, for each application class, we analyze the performance ranking of all feasible strategies. As a result, for each class, we have a set of suitable partitioning strategies, and their ranking in terms of performance.

To evaluate the proposed partitioning strategies and the correctness of the performance ranking, we select six representative applications from different application classes [115]. Our experiments show that, for each application, we are able to choose the best performing strategy, and the theoretical performance ranking we propose matches the empirical ranking. By applying the best partitioning on these applications, we achieve on average 3.0×/5.3× speedup compared to the Only-GPU/CPU execution.

In summary, *the main contributions* in this chapter are: (1) we propose a classification of data parallel applications, making it possible to design suitable partitioning strategies for different application classes; (2) we extend the usability of static partitioning, and further propose a mix of static and dynamic partitioning strategies to achieve both wide applicability and high performance for our workload partitioning; (3) we determine the performance-based ranking of all suitable partitioning strategies for each application class (we also validate the ranking empirically); (4) we design an application analyzer that selects the best performing partitioning strategy for a given application, leading to efficient

application execution on heterogeneous platforms.

The rest of the chapter is organized as follows. We briefly discuss existing successful partitioning solutions in Section 6.1. In Section 6.2, we present our analyzer with the application classification and the proposal of partitioning strategies. The empirical evaluation is given in Section 6.3, followed by a discussion in Section 6.4. We study related work in Section 6.5, and summarize this chapter in Section 6.6.

6.1 Background

In this section, we discuss the usage of static partitioning and dynamic partitioning on heterogeneous platforms. Based on this introduction, we analyze the strengths and limitations of each partitioning solution and the motivation for combining them for high performance heterogeneous computing.

6.1.1 Our Static Partitioning Approach

In Chapter 5, we have generalized a static partitioning approach that predicts the optimal workload partitioning for single-kernel applications running on heterogeneous platforms. This approach supports various platforms, with one or more accelerators, identical or non-identical. Figure 6.1 shows an overview of the approach.

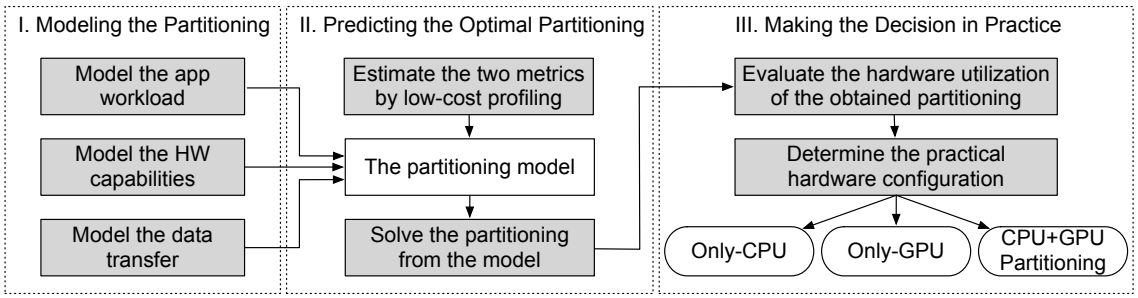


Figure 6.1: Our static partitioning approach.

The approach consists of three main steps. The first step is *modeling the partitioning*, where we model the execution of the workload on the heterogeneous platform. Given a fitting criteria (e.g., the minimum execution time), we build a partitioning model that represents the optimal partitioning. This partitioning model integrates the modeling of application workload, hardware capabilities, and data transfer overhead together, and is finally expressed as an equation with two derived metrics—(1) the relative hardware capability (the ratio of GPU throughput to CPU throughput), and (2) the GPU computation to data transfer gap (the ratio of GPU throughput to data-transfer bandwidth). The two metrics are the key factors for determining the optimal partitioning, and vary depending

on the platform to be used, and the application and the dataset to be computed. Next, for *predicting the optimal partitioning*, we use a low-cost profiling to estimate the values of the two metrics, ensuring a realistic estimation adaptive to any changes of platforms, applications, and datasets. By substituting the estimations into the partitioning model, we predict the optimal partitioning. The final step is *making the decision in practice*. This step determines the right hardware configuration (Only-CPU, Only-GPU, or CPU+GPU with workload partitioning) taking the actual hardware utilization into consideration. By checking if the obtained partitioning is able to efficiently use a certain amount of hardware cores of each processor, we take the decision to use either a single processor or the mix of both.

In summary, our static partitioning approach provides a systematic way to determine (1) the best performing hardware configuration and, when needed, (2) the optimal workload partitioning.

6.1.2 The OmpSs Programming Model

OmpSs [27] is a high level, task-based programming model that supports heterogeneous many-core systems. In OmpSs, a *task* is an independent, parallelizable section of code that corresponds to a kernel. The user annotates a kernel with the *task* construct, and defines (1) the task size and (2) the task data dependencies. When the program execution reaches a task annotation, the OmpSs runtime will create an instance of the task, and will schedule it to a compute resource (e.g., a CPU core, a GPU) for execution. The total number of task instances is proportional to the user-defined task size.

OmpSs uses a *thread-pool* execution model [11], similar to the OpenMP execution model, to support heterogeneous and asynchronous task execution. The OmpSs runtime creates a team of software threads, including SMP threads and accelerator helper threads. In OmpSs these threads are created at program startup, while in OpenMP they are usually created when the program reaches the first parallel region.

A thread manages the execution of task instances on a compute resource. The runtime analyzes data dependencies (indicated by the user) of each created task instance, adds it to a *task dependency graph*, and schedules it when all its dependencies are fulfilled and there is a free compute resource. This ensures a correct, asynchronous execution of tasks, i.e., the team of threads can process different task instances (from the same kernel and/or different kernels) concurrently. The *taskwait* construct provides a way to set a global synchronization point after a kernel. At this point, the program waits until all previously created task instances have completed.

OmpSs enables heterogeneous computing by using the *target* construct with several clauses. The *device* clause specifies which kind of device (i.e., compute resource) executes the task. Examples of devices are CPU cores (annotated as *smp*) and GPUs (anno-

tated as *opencl* or *cuda*). When using CPU cores, multiple SMP threads manage multiple task instances running on different cores. When using GPUs, the GPU helper threads execute the CUDA or OpenCL code using the native support libraries. The OmpSs memory model supports multiple memory spaces, and the runtime ensures data consistency (i.e., manages the data transfers) between memories by analyzing the user-defined data dependencies, and ensuring that before a task instance is executed in a memory space, data has been copied to it. Additionally, in heterogeneous environments, the taskwait construct not only synchronizes task execution but also flushes data in different memories to the host (the CPU) memory. The *implements* clause allows for multiple implementations of the same task for different kinds of compute resources, so the runtime can dynamically schedule a task instance to a compute resource based on a given scheduling policy. As a result, the OmpSs runtime dynamically partitions[1] the application workload to use different components of the heterogeneous platform.

6.1.3 Strengths and Limitations

Our approach uses a static partitioning strategy to determine the best performing hardware configuration and workload partitioning, but its usability is limited to single-kernel applications. This limitation is due to the foundation of the partitioning model, where the optimal partitioning ensures a perfect execution overlap between processors. To achieve such an overlap, the model needs to know explicitly in execution flow when the parallelism starts and ends. Single-kernel applications fit this requirement.

OmpSs uses a dynamic partitioning strategy, where the partitioning is determined at runtime without the need to know when a kernel starts and ends. It supports both single-kernel and multi-kernel applications. The runtime keeps correct data dependencies, enabling multi-kernel asynchronous execution and inter-kernel parallelism. However, dynamic partitioning introduces runtime scheduling overhead (including multiple data transfers on heterogeneous platforms) which does not exist in static partitioning. In addition, the application performance largely depends on the choice of scheduling policy and task size. A wrong scheduling policy or an unwise task size selection can lead to suboptimal partitioning and degraded performance.

Observing that both single-kernel and multi-kernel applications exist, and each partitioning strategy (static or dynamic) has its strengths and limitations, it is necessary to design an application analyzer that proposes the most suitable partitioning strategy for each type of applications. This is the goal of this work.

[1] Usually, dynamic scheduling is used when referring to the case of multi-kernel applications, and dynamic partitioning for single-kernel applications. In this chapter, we use dynamic partitioning to refer to both cases.

6.2 The Application Analyzer

In this section, we present our application analyzer. We first present its requirements and overview. Next, we explain in detail the key design components: a classification of applications and a set of suitable partitioning strategies.

6.2.1 Requirements

Firstly, we need to define and classify applications according to the *application kernel structure*. The kernel structure shows the number of kernels an application has and the kernel execution flow. Secondly, for each application class, we need to propose feasible partitioning strategies and select the one that best improves application performance.

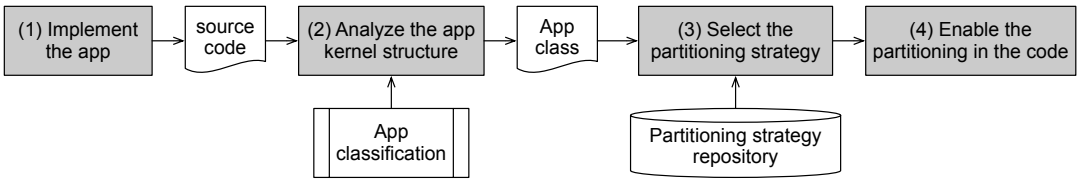


Figure 6.2: The application analyzer overview.

Figure 6.2 gives an overview of our application analyzer from input to output. (1) We parallelize the application, obtaining the source code. (2) From the source code, we analyze the application kernel structure, and identify the class the application belongs to. (3) We select the best performing partitioning strategy from a set of options based on the determined application class. (4) According to the decision, we enable the corresponding partitioning strategy in the source code. In our work, we use OmpSs to parallelize the application, because it enables not only dynamic partitioning but also static partitioning with only few modifications. We note that the use of our analyzer is not limited to OmpSs, and users can apply our analyzer to their own implementations.

6.2.2 Application Classification

We use the application kernel structure to classify applications. Specifically, we use two criteria: the number of kernels and the type of kernel execution flow, which can be a sequence (one kernel after another), a loop, or a full DAG (kernel execution forming a directed acyclic graph). We propose therefore five application classes, shown in Figure 6.3.

- **SK-One** (Class I) only has a single kernel.
- **SK-Loop** (Class II) also has a single kernel, but the kernel is iterated in a loop.

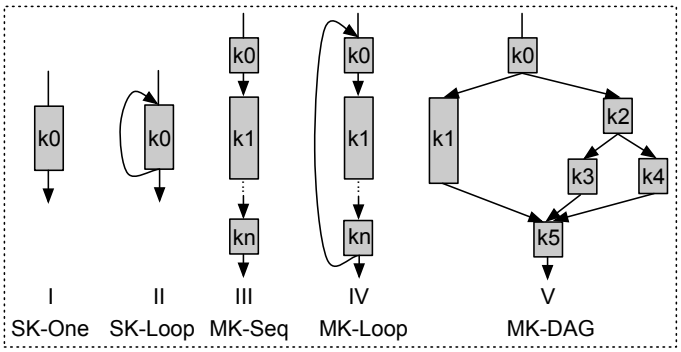


Figure 6.3: Application classification results and the typical kernel structure of each application class. Each rectangle represents a kernel, and its length represents the amount of computation the kernel needs to perform. The arrows show the kernel execution flow.

- **MK-Seq** (Class III) has multiple kernels of different kinds, and the kernel execution follows a sequence.
- **MK-Loop** (Class IV) has multiple kernels of different kinds. The kernels are executed in a sequence, and the sequence is further iterated in a loop.
- **MK-DAG** (Class V) has multiple kernels of different kinds. The kernel execution forms a DAG, and therefore the execution is expected to be very dynamic.

To determine the coverage of applications by these five classes, we have examined five benchmark suites (a total of 86 applications) [115]. The study shows that the five classes cover all 86 applications. In Classes III–V, there are some applications in which one or some of kernels are iterated in a loop. The actual execution of that loop can be unfolded/unrolled. Therefore, the loop structure does not affect the application's main kernel structure and the partitioning strategies that can be applied.

6.2.3 Partitioning Strategies

We propose five partitioning strategies suitable for different classes of applications and execution scenarios (illustrated in Figure 6.4). The five strategies are proposed starting from existing successful partitioning solutions with minimal changes, and provide more than sufficient coverage for each application class. As our and the other previous static partitioning solutions only work for single-kernel applications, we extend the applicability of static partitioning to multi-kernel applications.

- **SP-Single** is a static partitioning strategy, based, for example, on our approach (see Section 6.1.1), to determine a static partitioning for a single kernel. It is used for the SK-One and SK-Loop classes. We assume that, for SK-Loop, the kernel has

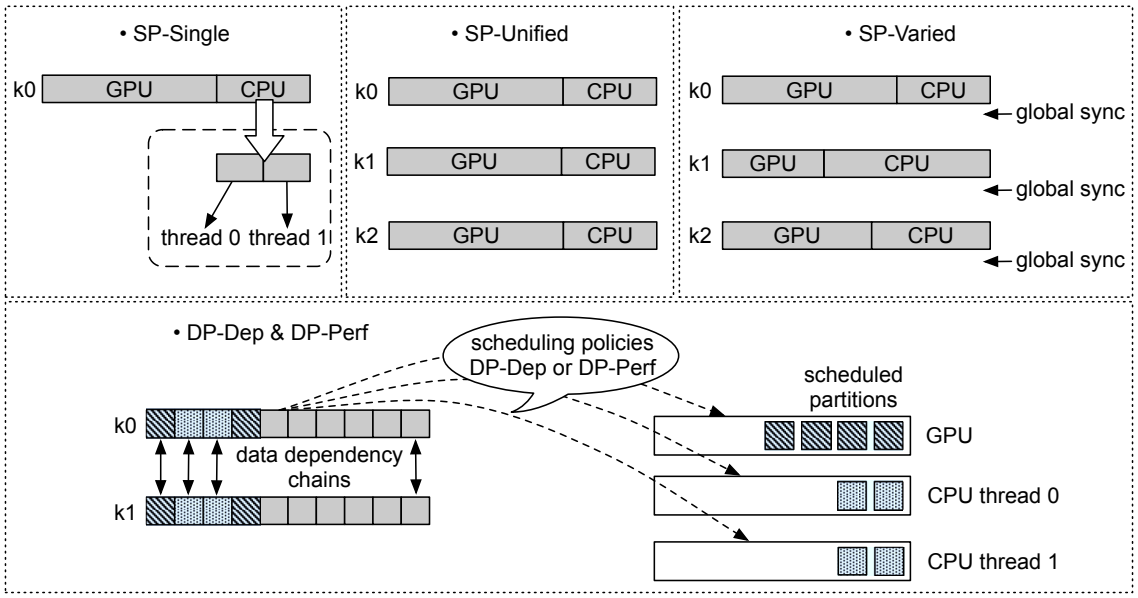


Figure 6.4: The proposed partitioning strategies. Each kernel is represented by a rectangle. The horizontal width represents the kernel's problem size (the vertical length that shows the amount of computation is omitted for simplicity). In SP-Single, the whole problem is partitioned into a GPU task and a CPU task. When using OmpSs, the CPU task is actually partitioned into multiple CPU task instances managed by multiple threads (e.g., two task instances assuming we have two threads). In SP-Unified, all the kernels have the same partitioning point. In SP-Varied, different kernels have different partitioning points. In DP-Dep and DP-Perf, each kernel is partitioned into multiple task instances, and each task instance is scheduled to run on either the GPU or the CPU (each CPU core in OmpSs) based on the scheduling policy.

stable performance in the loop, and therefore the partitioning remains the same (i.e., we determine the partitioning for one iteration, and use it for all iterations). If this assumption is not true, we can regard each iteration of the kernel as a different kernel, thus turning a SK-Loop application into a MK-Seq application.

- **SP-Unified** is designed for MK-Seq and MK-Loop based on the SP-Single partitioning strategy. In SP-Unified, we regard all the kernels as a single, fused kernel, and determine a unified partitioning. The condition to use this strategy is that the application does not need global synchronization between two consecutive kernels (e.g., a *taskwait* construct in OmpSs). Each device processes its partition (of each kernel) without inter-kernel synchronization with the host. As a result, the data locality is preserved on each device, and there is only a single host-to-device data transfer before the first kernel starts, and one device-to-host data transfer after the last kernel finishes.

- **SP-Varied** is also designed for MK-Seq and MK-Loop, where we apply the SP-Single partitioning strategy kernel by kernel, resulting in varied partitioning points. SP-Varied is used for the cases where applications need inter-kernel synchronization: (1) applications originally use synchronization to flush the data to the host for post-processing, or (2) due to the partitioning, applications need synchronization to assemble the output data of one kernel produced on different processors for the correct input of the next kernel. The use of global synchronization incurs multiple data transfers between the host and the devices.

- **DP-Dep** is a dynamic partitioning strategy usable for all application types. It schedules partitions (task instances) to devices in a breadth first order. If the application falls in Classes II–V, DP-Dep keeps tracking the data dependency chain to assign partitions that belong to the same chain to the same device, minimizing the data transfers.

- **DP-Perf** is a dynamic partitioning strategy also usable for all application types. It also tracks data dependency as DP-Dep, and implements a performance-aware scheduling policy. For each kernel, the runtime profiles how fast each device processes a partition, and estimates the device busy time (when a device becomes free for use as it finishes all the partitions that have been already assigned to it). The information is kept and updated to determine which device will be the earliest executor for the next coming partition, and the runtime will schedule the coming partition to that device. DP-Dep and DP-Perf are provided by the OmpSs runtime, and further details can be found in [102].

Next, we summarize suitable partitioning strategies for each application class in Table 6.1, and theoretically analyze their performance ranking.

Table 6.1: Suitable partitioning strategies for each application class.

Application classes	Suitable strategies and performance ranking
SK-One, SK-Loop	1. SP-Single 2. DP-Perf 3. DP-Dep
MK-Seq, MK-Loop (w/o sync)	1. SP-Unified 2. DP-Perf 3. DP-Dep 4. SP-Varied
MK-Seq, MK-Loop (w sync)	1. SP-Varied 2. DP-Perf 3. DP-Dep 4. SP-Unified
MK-DAG	1. DP-Perf 2. DP-Dep

Proposition 1: For *all classes*, DP-Perf $\geq$ DP-Dep (where "$\geq$" means "outperforms or equals").

Discussion: DP-Perf uses performance information to determine the scheduling. It is able to distinguish different devices and balance the workload. DP-Dep cannot distinguish such a difference, and may assign too much work to one device, leading to worse

performance compared to DP-Perf. This proposition shows that it is necessary to apply a performance-aware scheduling policy when using dynamic partitioning on heterogeneous platforms.

Proposition 2: For *SK-One* and *SK-Loop*, SP-Single $>$ DP-Perf $\geq$ DP-Dep.

Discussion: SP-Single determines the optimal partitioning, while DP-Perf and DP-Dep cannot guarantee the optimality all the time. Ideally, a performance-aware partitioning, like DP-Perf, detects the same partitioning as SP-Single, but it still cannot outperform SP-single due to the scheduling overhead. According to Proposition 1, the second part of the inequality holds as well.

Proposition 3: For *MK-Seq* and *MK-Loop*, (1) if the application does not need inter-kernel synchronization, SP-Unified $>$ DP-Perf $\geq$ DP-Dep $\geq$ SP-Varied; (2) if the application originally uses or needs (due to the partitioning) inter-kernel synchronization, SP-Varied $>$ DP-Perf $\geq$ DP-Dep $\geq$ SP-Unified.

Discussion: (1) For the application that does not need inter-kernel synchronization, SP-Unified regards all the kernels as a single kernel, and determines a single, unified partitioning. According to Proposition 2, it outperforms DP-Perf and DP-Dep. SP-Varied regards each kernel as an independent kernel, and determines the partitioning separately. To enable SP-Varied, we need to know explicitly when each kernel starts and ends, which means we need to add extra global synchronization points between kernels. This leads to multiple inter-kernel data transfers, an unnecessary, expensive overhead for the application. Therefore, SP-Varied performs the worst. (2) For the application that originally uses or needs (due to the partitioning) inter-kernel synchronization, the execution flow is segmented by the synchronization points. SP-Varied ensures the optimal partitioning for each kernel, so it ensures the overall best performance that none of the other strategies can achieve. SP-Unified fixes a unified partitioning regardless of kernel differences, so it may result in severe workload imbalance and worse performance compared to DP-Perf or even DP-Dep.

For *MK-DAG*, because the execution flow is too dynamic, feasible partitioning strategies are DP-Perf and DP-Dep. It may be possible to apply static partitioning to certain kernel(s), but this requires adding extra synchronization point(s), and may or may not bring in performance improvement (which is application-specific). Thus, we recommend the use of dynamic partitioning strategies for this class.

6.3 Experimental Evaluation

In this section, we present the evaluation of the proposed partitioning strategies. We select a set of applications, and for each application, we test all partitioning strategies, compare their performance, and select the best performing one. Next, we verify that the empirical ranking indeed fits with the theoretical ranking we have proposed in Section 6.2.3.

In addition, we also measure the performance of Only-CPU and Only-GPU executions[2] to see how much performance is gained by using the best partitioning strategy on the heterogeneous platform.

We note that for applications in the MK-DAG class, we cannot obtain valuable performance comparison between static and dynamic partitioning strategies, so we exclude the experiments for this class[3].

6.3.1 Experimental Setup

Applications

We present in Table 6.2 the applications we used for all the experiments. We note that all these applications are selected after our study in [115], and are representative for the classes of applications we focus on. MatrixMul is a dense matrix-matrix multiplication. BlackScholes is a financial model to calculate European option prices. These two applications fall in the SK-One class, as they have a single kernel for parallelization. Nbody is a scientific application that simulates interactions of individual bodies over time. HotSpot is a thermal modeling application that computes the temperature of a grid of cells over time. These two applications belong to the SK-Loop class, because both of them execute a single kernel iteratively. The STREAM benchmark is used to test memory bandwidth with four different kernels. We use STREAM in two forms: STREAM-Loop (the original form) executes the four kernels multiple times, and STREAM-Seq executes the four kernels one time (by limiting the number of iterations). They fall in the MK-Loop class and the MK-Seq class, respectively.

Table 6.2: Applications for evaluation.

Application	Application class	Origin
MatrixMul	SK-One	Nvidia OpenCL SDK [91]
BlackScholes	SK-One	Nvidia OpenCL SDK
Nbody	SK-Loop	Mont-Blanc benchmark suite*
HotSpot	SK-Loop	Rodinia benchmark suite [17]
STREAM-Seq	MK-Seq	The STREAM benchmark [84]
STREAM-Loop	MK-Loop	The STREAM benchmark

*Mont-Blanc is implemented in OmpSs by Barcelona Supercomputing Center.

[2]Only-CPU is the parallel execution that only uses the CPU. Only-GPU is the parallel execution that only uses the GPU.

[3]We refer interested readers for the performance comparison of the two dynamic partitioning strategies in [102].

Implementation

The selected applications originate from different benchmarks, so we port the code[4] to use OmpSs. The implementation can be summarized in two steps.

In the *task implementation* step, we prepare, for each kernel, a CPU implementation and a GPU implementation. The CPU implementation is the sequential implementation, and the GPU implementation is the kernel in OpenCL or CUDA (we use OpenCL in this work). By using the *task* and *target* constructs in OmpSs, we annotate the two implementations as the CPU task and the GPU task, respectively.

In the *partitioning implementation* step, we prepare two code versions: one for static partitioning and one for dynamic partitioning. In static partitioning, the GPU task is invoked once, and the CPU task is invoked m times to create m task instances mapping to m threads. Assuming n is the full problem size, and n_g and n_c are the GPU and CPU problem sizes obtained using static partitioning strategies ($n = n_g + n_c$), we set the GPU task size as n_g, and the CPU task size as n_c/m. In dynamic partitioning, we use the *implements* clause to notify the runtime that the GPU task and the CPU task are two implementations of the same task. To create m task instances, we need to set the task size as n/m. The runtime decides to which device each task instance is scheduled.

Hardware and Software

We perform our evaluation on a heterogeneous platform which integrates an Intel Xeon E5-2620 CPU (Hyper-Threading enabled) and an Nvidia Tesla K20 GPU. Table 6.3 lists the hardware information.

Table 6.3: The hardware platform.

	CPU	GPU
Processor	Intel Xeon E5-2620	Nvidia K20m
Frequency (GHz)	2.0	0.705
#Cores	6 (12 as HT enabled)	2496 (13 SMXs)
Peak GFLOPS (SP/DP)	384.0/192.0	3519.3/1173.1
L1 cache size (KB)	192	208
L2 cache size (KB)	1536	1536
L3 cache size (MB)	15	No L3 cache
Memory capacity (GB)	64	5
Peak Memory Bandwidth (GB/s)	42.6	208.0

The OmpSs version we use is OmpSs-14.10 (runtime version 0.7.4, compiler version 1.99.4). The applications are compiled with OmpSs compiler with -O3 option. The back-

[4]Code available at http://www.pds.ewi.tudelft.nl/jieshen/code/.

end compilers for the CPU and the GPU are Intel ICC 13.3 and Nvidia OpenCL 5.5, respectively.

6.3.2 Performance Evaluation

This subsection presents the performance evaluation per application class. To make the experiments consistent, we set the same number of threads (m) for all the execution scenarios that use the CPU (i.e., Only-CPU, static partitioning, and dynamic partitioning). We vary m to be a multiple of CPU cores in Only-CPU, and use the best-performing one. The task size in Only-CPU (n/m), static partitioning (n_c/m), and dynamic partitioning (n/m) are determined accordingly. When using static partitioning strategies, we profile the CPU and the GPU to estimate the hardware capabilities, and use the estimation to predict the optimal partitioning (see Section 6.1.1). When using the DP-Perf strategy, there is a fixed profiling phase where each device gets 3 task instances to make the runtime learn each device's performance. Both profiling phases are excluded from the performance comparison.

The SK-One Class

We evaluate MatrixMul and BlackScholes for this class. MatrixMul performs a dense matrix-matrix multiplication (A $\times$ B = C). The matrix size we use is 6144$\times$6144 (0.4 GB). We apply row-wise partitioning, i.e., each task instance receives multiple consecutive rows of A and the full B, and performs the computation for corresponding rows of C. BlackScholes performs a series of floating point operations to calculate the option prices. The options are stored in a 1D array. We use 80,530,632 options (1.5 GB) for evaluation, and each task instance receives a number of neighboring options.

The partitioning strategies we compare are SP-Single, DP-Perf, and DP-Dep. Figure 6.5 presents the performance results. We also calculate the partitioning ratio of different strategies, and present it in Figure 6.6. For dynamic partitioning, we count the number of task instances assigned to the CPU and the GPU, respectively, and convert it to the ratio.

In MatrixMul, we see that Only-GPU performs much better than Only-CPU. This is because MatrixMul is a compute-intensive application suitable for GPU acceleration, and the GPU data transfer overhead only takes a small proportion of the GPU execution time. The SP-Single strategy detects this fact, and decides to assign approximately 90% of the data to the GPU and the remaining 10% to the CPU[5], leading to the best performance. As most of the work is on the GPU, the performance of SP-Single is close to that of Only-GPU. DP-Perf uses performance-aware scheduling. It actually assigns all the task

[5]In static partitioning, the final n_g is rounded up to a multiple of GPU warp size, and the final n_c is calculated as $n - n_g$.

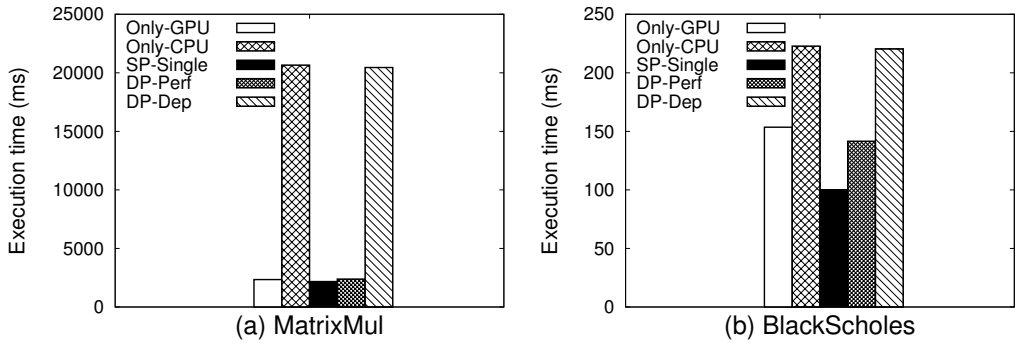


Figure 6.5: The execution time (ms) of different strategies in SK-One.

instances to the GPU, so it performs slightly worse than SP-Single (i.e., DP-Perf overestimates the GPU capability). DP-Dep performs much worse than DP-Perf and SP-Single, because DP-Dep by default uses all the CPU cores and the GPU, but does not take into account their different hardware capabilities. As a result, only one task instance is assigned to the GPU and the rest to the CPU, leading to workload imbalance.

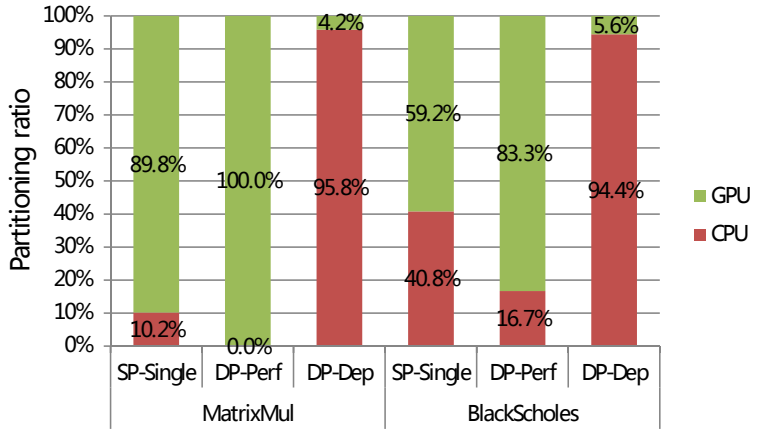


Figure 6.6: The partitioning ratio of different strategies in SK-One.

We see similar performance behavior in BlackScholes. SP-Single performs the best out of all. This application has a large data transfer overhead (the data transfer takes 37.5× more time than the GPU kernel computation), and SP-Single calculates a 41%/59% assignment to the CPU/GPU heterogeneous platform. DP-Perf also detects the performance change but still overestimates the GPU capability. As a result, the number of task instances assigned to the GPU exceeds the optimal, degrading the overall performance. Again, DP-Dep performs the worst because it assigns too much work to the CPU.

The SK-Loop Class

We evaluate Nbody and Hotspot for this class. Both applications perform a kernel computation in a loop, and the computation output of one iteration is the input of the next iteration. There is a global synchronization point after each iteration to ensure the outputs from different processors are combined at the host and updated to the input buffer before the next iteration. In Nbody, we compute the status of 1,048,576 bodies stored in 1D arrays (64 MB). In HotSpot, the grid size is 8192×8192 (0.75 GB), and we apply row-wise partitioning as for MatrixMul.

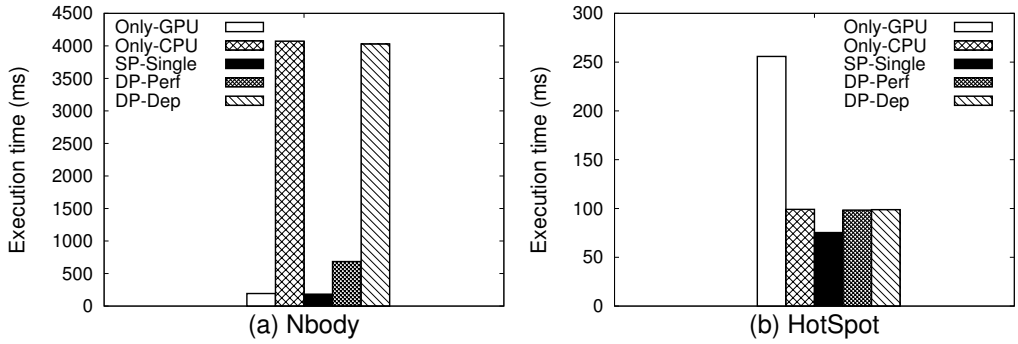


Figure 6.7: The execution time (ms) of different strategies in SK-Loop.

The partitioning strategies we compare are SP-Single, DP-Perf, and DP-Dep. In SP-Single, we determine the partitioning for one iteration, and use it for all iterations. Figure 6.7 shows the performance comparison, and Figure 6.8 shows the partitioning ratio. We see that in both applications, SP-Single gets the best performance. For Nbody, the GPU performs much better than the CPU, so SP-Single assigns most of the work to the GPU. On the contrary, HotSpot has better performance on the CPU, and SP-Single assigns a large partition to the CPU (the GPU performs worse mainly due to the data transfer overhead). DP-Perf detects similar partitioning as SP-Single in both applications, but its performance is worse than SP-Single (even worse than Only-GPU in Nbody) because of the dynamic partitioning overhead, which includes multiple times of taking scheduling decisions, OpenCL kernel invocations, and data transfers. DP-Dep does not distinguish different processors, thus resulting in the worst performance.

Summary 1: SP-Single is the best performing partitioning strategy for applications in the SK-One and SK-Loop classes. If we choose DP-Perf or DP-Dep, we observe sub-optimal performance.

The MK-Seq Class

We evaluate STREAM-Seq for this class. The application performs 4 different kernels (copy, scale, add, and triad) on 1D arrays. The number of array elements is 62,914,560

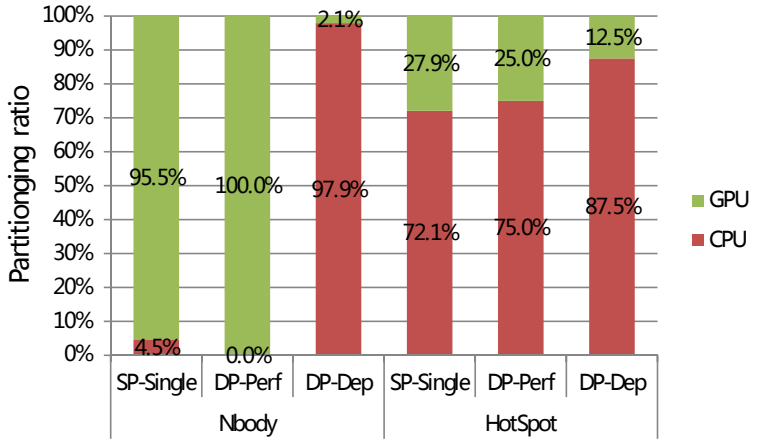


Figure 6.8: The partitioning ratio of different strategies in SK-Loop.

(0.7 GB). We further consider two cases of executions, with and without inter-kernel synchronization. We note that inter-kernel synchronization is not necessary for this application, but we manually add it to mimic applications that need synchronization (see Section 6.2.3).

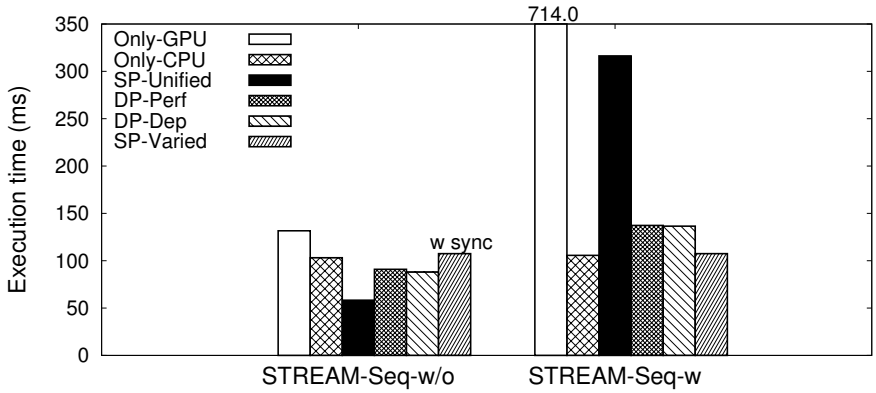


Figure 6.9: The execution time (ms) of different strategies in MK-Seq. "w" and "w/o" represent the execution with and without inter-kernel synchronization, respectively. SP-Varied in the two cases are the same, because using SP-Varied requires inter-kernel synchronization.

SP-Unified, SP-Varied, DP-Perf, and DP-Dep are the partitioning strategies to be considered. Figure 6.9 shows the performance comparison results. When there is no inter-kernel synchronization, SP-Unified performs the best. It regards all the kernels as a single, fused kernel, and keeps the partitioning at 44% of the elements on the GPU and 56% of the elements on the CPU. The GPU gets less work mainly because its data transfer takes too much time (around 88% of the overall execution time). DP-Perf and DP-Dep

rank second. Both strategies allow for asynchronous executions of task instances from different kernels. We note that there is no visible performance difference between the two strategies. This is because DP-Dep assigns most of the task instances to the CPU, which coincidentally matches the partitioning obtained by DP-Perf. SP-Varied performs the worst, because the use of this strategy requires extra global synchronization points, leading to extra data transfers between kernels. Compared to SP-Unified, the partitioning is skewed towards the CPU in SP-Varied. Figure 6.10 shows the partitioning ratio of different strategies.

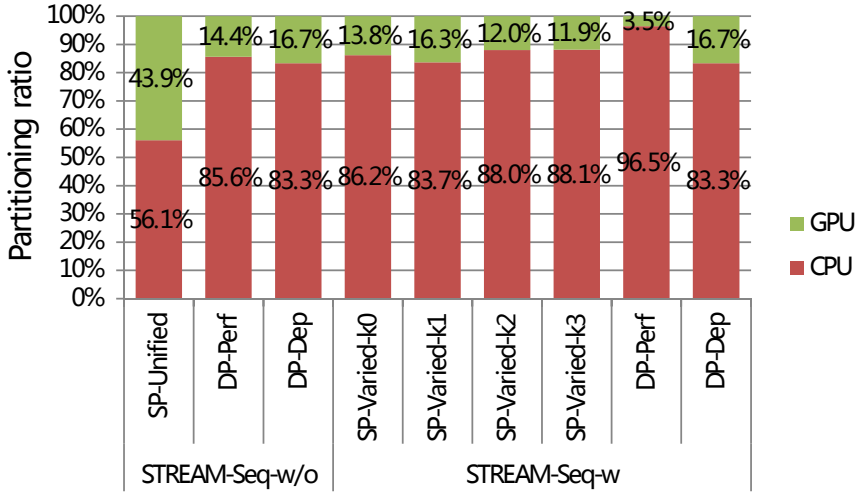


Figure 6.10: The partitioning ratio of different strategies in MK-Seq. The ratios of SP-Varied are the same in both cases and presented per kernel only in STREAM-Seq-w.

When there is inter-kernel synchronization, SP-Varied becomes the best performing strategy, as it varies the partitioning per kernel, and ensures each kernel gets the best performance. DP-Perf and DP-Dep still rank second. In dynamic partitioning, the synchronization serializes the kernel execution flow, leading to 35% performance degradation compared to that without synchronization. In SP-Unified, we use the partitioning obtained in the case without synchronization. This partitioning makes the GPU get too much work for each kernel, and therefore gets the worst performance.

The MK-Loop Class

We evaluate STREAM-Loop for this class. Similar to STREAM-Seq, we compare the execution with and without inter-kernel synchronization. The problem size is the same as that used in STREAM-Seq.

Figure 6.11 shows the performance comparison, and Figure 6.12 shows the partitioning ratio. When there is no inter-kernel synchronization, Only-GPU outperforms Only-CPU (different from STREAM-Seq), because the 4 kernels are iterated multiple times,

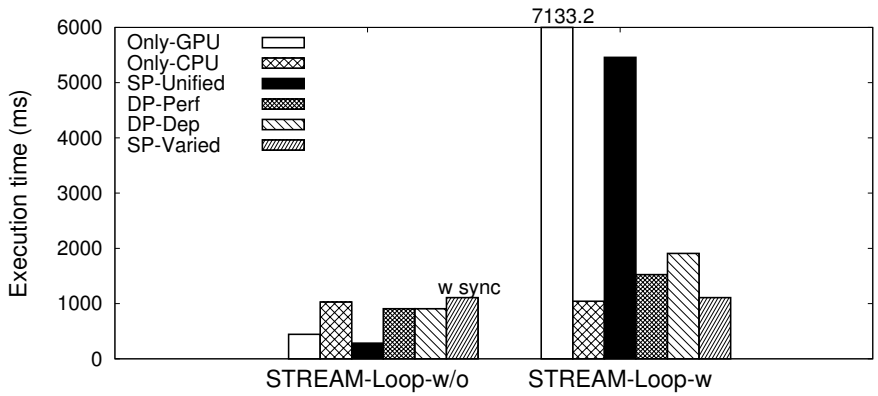


Figure 6.11: The execution time (ms) of different strategies in MK-Loop. "w" and "w/o" represent the execution with and without inter-kernel synchronization, respectively. SP-Varied in the two cases are the same, because using SP-Varied requires inter-kernel synchronization.

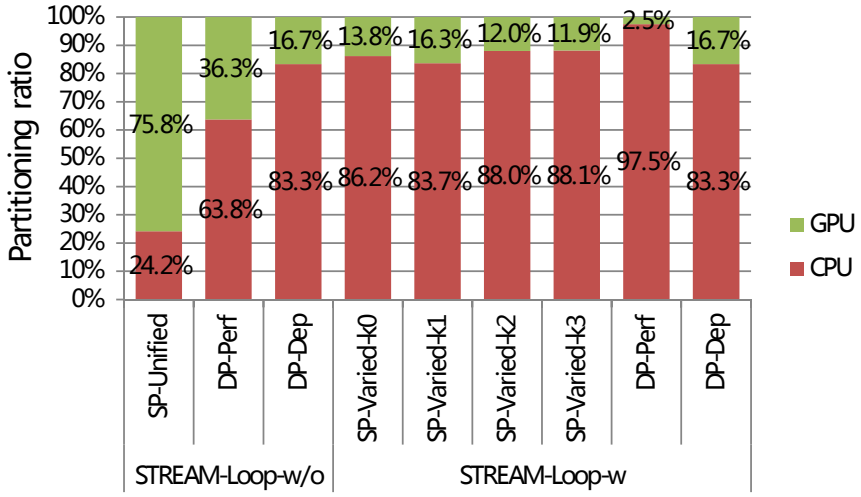


Figure 6.12: The partitioning ratio of different strategies in MK-Loop. The ratios of SP-Varied are the same in both cases and presented per kernel only in STREAM-Loop-w.

increasing the computation workload. As a result, more computation is assigned to the GPU. The other performance results are similar to those in STREAM-Seq: (1) SP-Unified obtains the best performance in the case without synchronization. We determine a unified partitioning for one iteration, and use it for all iterations. We note that the data transfer is not profiled, because all the iterations except the first and the last ones do not have any data transfer. (2) SP-Varied performs the best in the case with synchronization. As the partitioning is obtained by profiling one iteration, the partitioning ratio per kernel is the same as that in STREAM-Seq. (3) DP-Perf and DP-Dep take second place. The benefit of asynchronous execution (in the case without synchronization) increases as the number of iterations increases.

Summary 2: For applications in the MK-Seq and MK-Loop classes, the choice of the best partitioning strategy depends on whether the application needs inter-kernel synchronization: for applications that do not need synchronization, SP-Unified performs best; while for applications that need synchronization, SP-Varied performs best.

Overall Comments

The performance ranking of different partitioning strategies in our empirical evaluation matches the theoretical ranking we have proposed in Table 6.1.

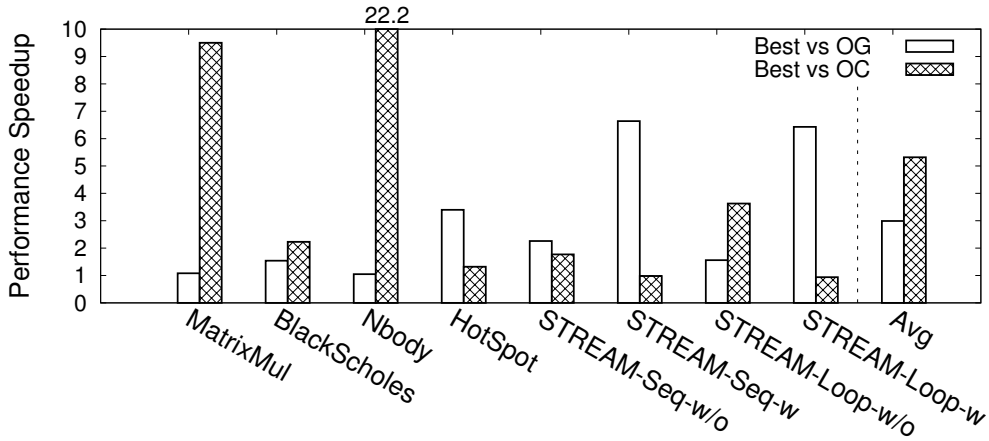


Figure 6.13: The speedup of the best partitioning strategy vs. Only-GPU (OG) and Only-CPU (OC) per application.

For each application, we compare the performance of the best partitioning strategy to that of the Only-GPU and Only-CPU executions, and present the speedup in Figure 6.13. We see that the performance improvement is application dependant. The speedup ranges from as much as 22× to close to 1×, mainly because each processor's capability is varied depending on the application. The average performance improvements for our six applications are 3.0× and 5.3× compared to Only-GPU and Only-CPU, respectively.

6.4 Discussion

From our experimental evaluation, we see that using partitioning on heterogeneous platforms improves application performance. The best performance is achieved by static partitioning strategies in 4 out of 5 application classes, as static partitioning, as long as it is applicable and optimal, ensures a perfect execution overlap between processors. Dynamic partitioning introduces scheduling overhead at runtime, and therefore gets less performance improvement even when it achieves the same partitioning ratio as static partitioning.

The scheduling policy has a significant influence on the performance of dynamic partitioning. A good policy on heterogeneous platforms should take into account the processors' difference in hardware capability. The task size (the granularity of partitioning) impacts performance as well. In our experiments, we have also varied the task size in dynamic partitioning, and find that the task size variation leads to performance variation. Thus, auto-tuning is recommended to find the best performing one. But even so, static partitioning outperforms dynamic partitioning for the first four classes of applications.

Pragmatically speaking, for a given parallel application which is not yet partitioned, we recommend using our application analyzer (see Section 6.2.1) to find the best partitioning strategy. If it is already partitioned in a dynamic way, but the best strategy is static partitioning (which is likely to happen), we can make dynamic partitioning "behave" like static partitioning: (1) set the task size to the full problem size, and determine the static partitioning ratio; (2) convert the static partitioning ratio to the task assignment ratio (e.g., k task instances on the CPU and l task instances on the GPU); (3) assign the determined numbers of task instances to the CPU and the GPU, respectively. Using this approach, the application gets a close-to-optimal partitioning with minimal manual effort.

6.5 Related Work

Static workload partitioning on CPU+GPU heterogeneous platforms has attracted quite some research recent years. Luk et al. proposed Qilin [78] which builds an analytical model based on curve-fitting to determine the best workload distribution. Insieme [67] uses offline training and machine learning to build a prediction model that derives the partitioning based on program features and problem-size related features. Grewe et al. applied a similar machine learning approach [37], and they also considered the partitioning in the presence of GPU contention [38]. SKMD from Lee et al. [71] utilizes a decision tree heuristic to search the best workload partitioning taking into account the performance variation of each processor. All these approaches determine a fixed, static partitioning between heterogeneous components of the platform, but their usability is limited to single-kernel applications. In this chapter, we extend the usability of static partitioning to multi-kernel applications.

Apart from that, single-kernel dynamic partitioning schemes have been developed to achieve load balancing at runtime. Boyer [14] proposed a scheduling algorithm based on varied chunk size. The chunk size is increased by a factor at each scheduling time, and the execution times of the scheduled chunks are used to partition the remaining work. Scogland et al. [108] proposed a runtime system that divides an accelerated OpenMP region across CPUs and GPUs based on four optional scheduling policies suitable for different execution scenarios. Ravi et al. [103] proposed both uniform- and non-uniform-chunk distribution schemes, where the latter assigns larger chunks to the GPU and smaller

chunks to the CPU. These schemes efficiently reduce scheduling overhead, but still cannot outperform the optimal partitioning determined by the static partitioning approaches.

Dynamic scheduling for multi-kernel applications has been proposed at two different scheduling granularities: per kernel and per chunk (of each kernel). Becchi et al [12] proposed a method to determine the location of each kernel taking the processor disparity and the data locality into consideration. The work in [39] determines the kernel-processor mapping based on code and runtime features. Unlike the OmpSs dynamic partitioning strategies [27, 102] used in this chapter, which schedules at chunk (task instance) granularity, these approaches cannot utilize inter-kernel parallelism to further improve application performance. StarPU [8] uses the same scheduling granularity as that in OmpSs, and therefore also enables inter-kernel asynchronous execution while maintaining correct data dependencies at runtime.

Compared to related work, in this chapter, we propose a set of five partitioning strategies, combining both static and dynamic features, for both single- and multi-kernel applications. Moreover, we propose an application classification and a performance ranking of different strategies for each application class. Thus, for a given application, we are able to determine the best strategy to partition the workload, maximizing the performance gain.

6.6 Summary

Workload partitioning is mandatory to improve application performance on heterogeneous platforms. To achieve both high performance and wide applicability for workload partitioning, matchmaking applications and partitioning strategies is desirable. In this chapter, we classify data parallel applications into five classes by analyzing the application kernel structure, and we propose a set of suitable partitioning strategies which combine static and dynamic approaches to cover all application classes. We further design an application analyzer that uses performance ranking to select the best performing strategy for a given application. Our work improves the applicability of static partitioning, and demonstrates its superiority, in many cases, over dynamic partitioning. By combining static and dynamic partitioning, our analyzer applies a unified method enabling a large variety of applications to be executed efficiently on heterogeneous platforms.

Chapter 7

Conclusions and Future Work

In this chapter, we summarize the findings of our research. Based on all these findings, we propose a generic framework to enable efficient utilization of heterogeneous platforms. We further outline directions for future research.

Heterogeneous platforms are becoming pervasive in high performance computing. Their efficient utilization is therefore becoming increasingly important. The heterogeneity of the hardware mix and the diversity of the applications pose significant challenges to exploiting such platforms. In this context, systematic methods to tackle these challenges are desirable.

This thesis studies how to systematically enable efficient high performance computing on heterogeneous platforms. We follow an application-centric approach, and we decompose data parallel applications into three dimensions—the parallelization dimension, the workload dimension, and the kernel dimension. From the parallelization dimension, we have understood how to parallelize applications using OpenCL, a unified programming model, on heterogeneous platforms (Chapter 2). Taking into account both the parallelization and the workload dimensions, we have studied efficient workload partitioning solutions for single-kernel balanced and imbalanced applications (Chapters 3, 4, 5). By considering all three dimensions together, we have enabled multi-kernel applications with complex kernel structures to benefit from workload partitioning over heterogeneous platforms (Chapter 6).

To summarize, this thesis demonstrates that *heterogeneous platforms* are the right solution, performance-wise, for many classes of data parallel applications, and shows how high performance can be achieved systematically.

7.1 Conclusions

Our research has led to the following conclusions.

1. **OpenCL is an efficient programming model for heterogeneous platforms under the assumption that users bear processor architectural difference in mind and tune their OpenCL code to fit different processors (RQ1).**

   Although a large collection of OpenCL code has been proved to behave well on GPUs, OpenCL's advantage of cross-platform portability should not be taken for granted when looking for well-performing OpenCL code on CPUs. Three categories of factors impact OpenCL performance on CPUs: (1) GPU-like programming style, (2) fine-grained parallelism, and (3) OpenCL compilers.

   We have shown how these factors can be addressed through simple, generic code transformations, which do not change the parallel structure of the application. These transformations can be, in principle, enabled or disabled by an auto-tuner, allowing OpenCL to use automated code specialization for different processors. This is as close as OpenCL can get to performance portability.

2. **Imbalanced applications fit naturally on heterogeneous platforms; but to *accelerate* imbalanced applications, workload partitioning is the key challenge (RQ2).**

   For an imbalanced application, a homogeneous platform, like a GPU or a CPU, might not be the right solution because the workload imbalance can heavily degrade hardware utilization.

   A heterogeneous platform enables task level parallelism for the imbalanced application, where the workload is partitioned into a CPU task and a GPU task with each task matching the usage pattern of its corresponding processor. For example, in the acoustic ray tracing case study used in our work, the relatively irregular part of the workload is mapped to the CPU for its fast sequential processing, and the relatively balanced part of the workload is mapped to the GPU for its massive parallelism. An auto-tuning of the task size is necessary to ensure partitioning optimality for any given application and platform.

3. **A model-based prediction method can be used for workload partitioning; the predictor preserves performance, and improves the efficiency of the partitioning process (RQ3).**

   The partitioning is a function of the processor capabilities as well as the application and the dataset to be used. To obtain the optimal partitioning that maximizes performance improvement, the processor capability difference, the application workload, and the CPU-GPU data transfer must all be included into the modeling.

   Characterizing the workload using a 2D workload model makes it possible to quantify the workload size and the workload imbalance. The estimation of the hardware

capabilities is achieved by building prior knowledge. In our work, we use profiling to build prior knowledge as it provides an realistic estimation by modeling the processor capabilities in the context of the application and dataset.

4. **Balanced applications can also be accelerated on heterogeneous platforms using a systematic approach to select the right hardware configuration and the optimal workload partitioning (RQ4).**

   Even for massively-parallel balanced applications, using GPUs alone might not be the best solution. It is often the case that the host, which is a multi-core CPU, is left unused. Its performance, combined with the negative impact of the CPU-GPU data transfer, make the CPU a valuable resource to use.

   Based on the prediction method developed for imbalanced applications, we have extended the modeling of workload partitioning to include balanced applications, and we have generalized a systematic approach that determines whether to use the GPU, the CPU, or both (with partitioning) for a given balanced application. We have further generalized this method to enable different profiling options suitable for different execution scenarios and to incorporate heterogeneous platforms with multiple accelerators.

5. **Static partitioning trades applicability for performance, while dynamic partitioning trades performance for applicability. To achieve both high performance and wide applicability, we need to combine them (RQ5).**

   Static partitioning is usually designed for single-kernel applications, but its use can be extended to certain classes of multi-kernel applications where the kernel execution follows a sequence or a loop. Dynamic partitioning cannot outperform static partitioning in these cases due to the scheduling overhead, but it can be used for the multi-kernel applications where the kernel execution is dynamic.

   We have shown that workload partitioning can be generalized to most classes of applications by following a three-stage process: (1) propose a structure-based classification of applications, (2) propose a set of static and dynamic partitioning strategies that cover all application classes, (3) design a matchmaking method to match applications and strategies based on a theoretically/empirically validated performance ranking. Specifically, we have proposed such a classification, and we have defined a simple matchmaking procedure that allows most data parallel applications to run efficiently on heterogeneous platforms.

7.2 Putting It All Together

In this thesis, we have developed a set of systematic methods to implement, partition, and map data parallel applications on heterogeneous platforms. We make one step further towards a unified system for efficient execution of parallel applications on heterogeneous platforms by combining all our findings into a single, generic framework. The design of this framework is presented in Figure 7.1. This is an update of the Glinda framework proposed in Chapter 3.

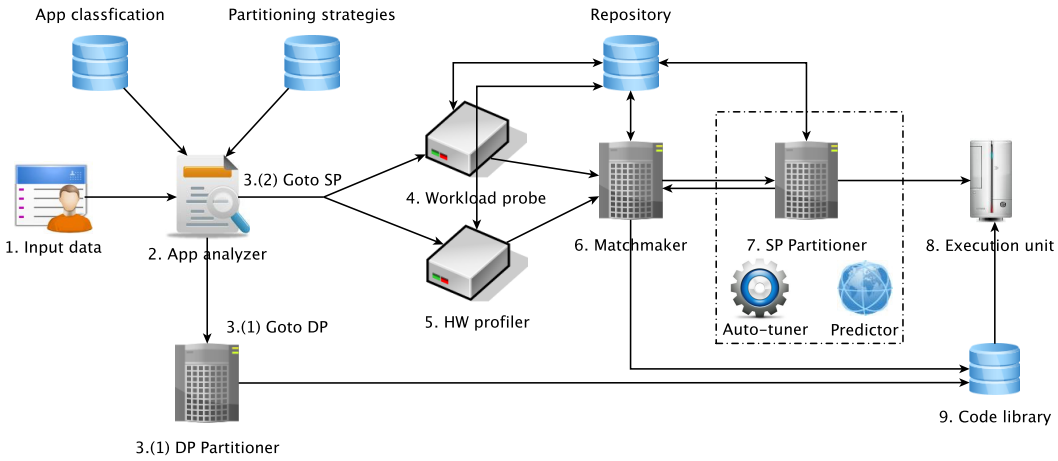


Figure 7.1: The Glinda 2.0 framework.

1. The *input data* for this framework includes the application (parallel, single-device code) and its parameters (if needed), together with the dataset. Ideally, this is the only information that the user needs to supply.

2. The *app analyzer* analyzes the source code, identifies the *application class* for the given application, and selects the best performing *partitioning strategy* that matches this class (Chapter 6).

3. (1) If a dynamic partitioning strategy is selected, the *DP partitioner* is invoked to partition the application workload into multiple chunks and schedule the chunks to processors on the *execution unit*, i.e., the target heterogeneous platforms (Chapter 6). (2) If a static partitioning strategy is selected, the *workload probe* and the *HW profiler* are invoked.

4. The *workload probe* characterizes the application workload (balanced or imbalanced), and generates the workload model (Chapters 4, 5).

5. The *HW profiler* profiles the processors to estimate the decisive partitioning metrics, i.e., the relative hardware capability and the GPU computation to data-transfer gap (Chapters 4, 5).

6. The *matchmaker* assumes that CPU+GPU is the best hardware configuration and asks the *SP partitioner* to determine the optimal partitioning (Chapter 5).

7. Either the *auto-tuner* or the *predictor* determines the partitioning and sends the obtained partitioning back to the matchmaker, where the practical hardware configuration (Only-CPU, Only-GPU, or CPU+GPU) is determined (Chapters 3, 4, 5). This approach works for single-kernel applications (Chapters 3, 4, 5), and we have extended it to a couple of classes of multi-kernel applications (Chapter 6). The results generated in 4–7 are stored in *repositories* for reuse.

8. The *execution unit* executes the application with the determined hardware configuration and workload partitioning, when needed (i.e., CPU+GPU is used).

9. Implementation-wise, single-device code (for single-processor execution) and multi-device code (for partitioned execution, in static and/or dynamic way) are implemented as code candidates in the *code library*. We recommend using OpenCL that expresses parallelism in a common code structure and allows for parameterized code specialization to tune the performance on each processor (Chapter 2).

Effectively, Glinda 2.0 enables a large variety of data parallel applications (with single or multiple kernels, with balanced or imbalanced workload) to achieve high performance on heterogeneous platforms.

7.3 Future Research Directions

Our work has provided a leap forward in the programmability of heterogeneous platforms. We discuss here a few immediate action points (1-3) that need to be addressed to transform the Glinda 2.0 design into a working framework, and we propose four alternative directions of research (4-7) for Glinda 3.0:

1. *Glinda 2.0 implementation.* While the concepts of Glinda 2.0 are quite clear, implementing this framework is not trivial. Research is needed for the app analyzer, the workload probe, and the HW profiler for making the framework fully automatic.

2. *Automatic code transformation.* This thesis has proposed generic code transformations to specialize OpenCL code for CPUs and GPUs. It is beneficial to implement the code transformations into an automated source-to-source compiler, making the

code specialization process transparent to users. Likewise, a compiler that automatically transforms single-device code to multi-device code is desirable for transparent workload partitioning.

3. *Porting Glinda 2.0 to more heterogeneous platforms.* Further effort can be made to port Glinda 2.0 to multiple types of heterogeneous platforms. For example, integrated CPU+GPU platforms are interesting options as their shared memory design makes it possible to eliminate data-transfer overhead and to address the GPU memory capacity limitation. In addition, a CPU with Intel MICs, with FPGAs, and heterogeneous mobile systems are popular platforms that can leverage workload partitioning to improve application performance.

4. *New benchmarking methodologies.* As heterogeneous computing is becoming popular, new benchmarking methodologies that help the community to better evaluate and improve heterogeneous computing frameworks are useful. Specifically, constructing multi-language applications with various classes of problems and datasets that exploit different patterns of computation and stress different hardware accelerators is a feasible approach, which in turn will lead to better framework design.

5. *Integration with distributed systems.* This thesis has proposed partitioning methods for single-node heterogeneous platforms. Integrating these methods into distributed systems equipped with heterogeneous nodes is an interesting research direction, as heterogeneity is becoming ubiquitous from parallel systems to distributed systems. Our methods, for intra-node workload partitioning, can be combined with existing inter-node workload scheduling policies to improve the overall application performance.

6. *Extension to different application classes.* Applications that exploit task level parallelism (e.g., streaming applications) and that exhibit dynamic runtime behavior (e.g., graph processing applications) can be too dynamic to model and to partition, as these applications' kernel executions tend to form a DAG. It is challenging to refine the classification of these applications, and investigate a finer-grained integration of intra-kernel static partitioning and inter-kernel dynamic scheduling. This research will further extend the usability of workload partitioning to more application classes beyond the applications addressed in this thesis.

7. *Extension of the partitioning model.* As a further step, the partitioning model can be extended to include other measurable fitting criteria. For example, as energy efficiency becomes increasingly important, using energy consumption as a fitting criterion to decompose the application workload will be useful for many heterogeneous systems with a limited energy budget.

Bibliography

[1] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In *Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'11)*, pages 932–943, 2011.

[2] A. Ali, U. Dastgeer, and C. Kessler. OpenCL for Programming Shared Memory Multicore CPUs. In *Proceedings of the 5th Workshop on MULTIPROG, in conjunction with HiPEAC 2012*, 2012.

[3] AMD. AMD Accelerated Parallel Processing OpenCL Programming Guide. http://developer.amd.com, 2012.

[4] AMD. APU 101: All about AMD Fusion Accelerated Processing Units. http://developer.amd.com/wordpress/media/2012/10/apu101.pdf, 2012.

[5] AMD. AMD OpenCL Optimization Guide. http://developer.amd.com/, 2014.

[6] M. Arntzen, S. A. Rizzi, H. G. Visser, and D. G. Simons. A framework for simulation of aircraft flyover noise through a non-standard atmosphere. In *the 18th AIAA/CEAS Aeroacoustics Conference*, pages AIAA–2012–2079, 2012.

[7] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[8] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[9] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Euro-Par 2009*, pages 851–862, 2009.

[10] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20:404–418, 2009.

[11] Barcelona Supercomputing Center. OmpSs Specification. http://pm.bsc.es/ompss-docs/specs/, 2015.

[12] M. Becchi, S. Byna, S. Cadambi, and S. T. Chakradhar. Data-Aware Scheduling of Legacy Kernels on Heterogeneous Platforms with Distributed Memory. In *SPAA 2010*, pages 82–91, 2010.

[13] L. Bergstrom. Measuring NUMA effects with the STREAM benchmark. *CoRR*, abs/1103.3225, 2011.

[14] M. Boyer, K. Skadron, S. Che, and N. Jayasena. Load Balancing in a Changing World: Dealing with Heterogeneity and Performance Variability. In *Computing Frontiers 2013*, pages 21:1–21:10, 2013.

[15] A. Branover, D. Foley, and M. Steinman. AMD Fusion APU: Llano. *IEEE Micro*, 32(2):28–37, 2012.

[16] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, 2013.

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC'09)*, pages 44–54, 2009.

[18] S. Che, J. W. Sheaffer, and K. Skadron. Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, pages 13:1–13:11, 2011.

[19] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? In *Proceedings of 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*, pages 225–236, 2010.

[20] A. Corrigan, F. Camelli, R. Löhner, and J. Wallin. Running Unstructured Grid-based CFD Solvers on Modern Graphics Hardware. *J. for Numerical Methods in Fluids*, 66(2):221–229, 2011.

[21] M. Daga, A. M. Aji, and W.-c. Feng. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In *2011 Symposium on Application Accelerators in High-Performance Computing (SAAHPC'11)*, pages 141–149. IEEE, 2011.

[22] M. Daga, T. Scogland, and W. Feng. Architecture-Aware Mapping and Optimization on a 1600-Core GPU. In *Proceedings of the 17th IEEE International Conference on Parallel and Distributed Systems (ICPADS'11)*, pages 316–323, 2011.

[23] S. Damaraju, G. Varghese, S. Jahagirdar, T. Khondker, R. Milstrey, S. Sarkar, S. Siers, I. Stolero, and A. Subbiah. A 22nm IA Multi-CPU and GPU System-on-Chip. In *2012 IEEE International Solid-State Circuits Conference (ISSCC'12)*, pages 56–57, 2012.

[24] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *GPGPU 2010*, pages 63–74, 2010.

[25] G. F. Diamos and S. Yalamanchili. Harmony: An Execution Model and Runtime for Heterogeneous Many core Systems. In *HPDC 2008*, pages 197–200, 2008.

[26] P. Du, R. Weber, P. Luszczek, S. Tomov, G. D. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming. *Parallel Computing*, 38(8):391–407, 2012.

[27] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[28] O. A. Fagerlund. Multi-core Programming with OpenCL: Performance and Portability - OpenCL in a Memory Bound Scenario. Master's thesis, Norwegian University of Science and Technology, 2010.

[29] J. Fang, A. L. Varbanescu, and H. J. Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. In *International Conference on Parallel Processing (ICPP'11)*, pages 216–225, 2011.

[30] R. Ferrer, J. Planas, P. Bellens, A. Duran, M. González, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta. Optimizing the Exploitation of Multicore Processors

and GPUs with OpenMP and OpenCL. In *Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC'10)*, pages 215–229, 2010.

[31] E. J. Fluhr, S. Baumgartner, D. W. Boerstler, J. F. Bulzacchelli, T. Diemoz, D. Dreps, G. English, J. Friedrich, A. Gattiker, T. Gloekler, et al. The 12-Core POWER8™ Processor With 7.6 Tb/s IO Bandwidth, Integrated Voltage Regulation, and Resonant Clocking. *IEEE Journal of Solid-State Circuits*, 50(1):10–23, 2015.

[32] G. Gan, X. Wang, J. Manzano, and G. R. Gao. Tile Percolation: An OpenMP Tile Aware Parallelization Technique for the Cyclops-64 Multicore Processor. In *Euro-Par 2009*, pages 839–850, 2009.

[33] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu. A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*, pages 345–354, 2012.

[34] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu. On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest. In *IPDPS 2013*, pages 851–862, 2013.

[35] C. Gregg, M. Boyer, K. Hazelwood, and K. Skadron. Dynamic Heterogeneous Scheduling Decisions Using Historical Runtime Data. In *ISCA Workshop 2011 (A4MMC)*, 2011.

[36] C. Gregg and K. M. Hazelwood. Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. In *ISPASS 2011*, pages 134–144, 2011.

[37] D. Grewe and M. F. P. O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *CC 2011*, pages 286–305, 2011.

[38] D. Grewe, Z. Wang, and M. F. P. O'Boyle. OpenCL Task Partitioning in the Presence of GPU Contention. In *LCPC 2013*, pages 87–101, 2013.

[39] D. Grewe, Z. Wang, and M. F. P. O'Boyle. Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. In *CGO 2013*, pages 1–10, 2013.

[40] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng. Twin Peaks: A Software Platform for Heterogeneous Computing on General-Purpose and Graphics Processors. In *PACT*, pages 205–216, 2010.

[41] T. Hamano, T. Endo, and S. Matsuoka. Power-aware Dynamic Task Scheduling for Heterogeneous Accelerated Clusters. In *IPDPS*, pages 1–8, 2009.

[42] P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing (HiPC'07)*, pages 197–208, 2007.

[43] Z. He and B. Hong. Dynamically Tuned Push-Relabel Algorithm for the Maximum Flow Problem on CPU-GPU-Hybrid Platforms. In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10)*, pages 1–10, 2010.

[44] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard. Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. In *Euro-Par 2010*, pages 235–246, 2010.

[45] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, 2008.

[46] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *ISCA 2009*, pages 152–163, 2009.

[47] S. Hong, T. Oguntebi, and K. Olukotun. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In *Proceedings of 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*, pages 78–88, 2011.

[48] Q. Hu, N. A. Gumerov, and R. Duraiswami. Scalable Fast Multipole Methods on Distributed Heterogeneous Architectures. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, pages 36:1–36:12, 2011.

[49] S. Huang, S. Xiao, and W. Feng. On the Energy Efficiency of Graphics Processing Units for Scientific Computing. In *Proceedings of the 23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09)*, pages 1–8, 2009.

[50] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. Stan. HotSpot: A Compact Thermal Modeling Methodology for Early-Stage VLSI Design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(5):501–513, 2006.

[51] Intel. Sandy Bridge. http://ark.intel.com.

[52] Intel. Auto vectorization of OpenCL code with the Intel SDK for OpenCL Applications. http://software.intel.com/en-us/articles/auto-vectorization-of-opencl-code-with-the-intel-opencl-sdk/, 2012.

[53] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com, 2012.

[54] Intel. Intel SDK for OpenCL Applications 2012 OpenCL Optimization Guide. http://software.intel.com/sites/landingpage/opencl/optimization-guide/index.htm, 2012.

[55] Intel. Intel SDK for OpenCL Applications 2012 User's Guide. http://software.intel.com/sites/landingpage/opencl/user-guide/index.htm, 2012.

[56] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU Communication Management and Optimization. In *PLDI 2011*, pages 142–151, 2011.

[57] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., 1st edition, 2013.

[58] H. Jia, Y. Zhang, G. Long, J. Xu, S. Yan, and Y. Li. GPURoofline: A Model for Guiding Performance Optimizations on GPUs. In *18th International Conference Euro-Par 2012 (EuroPar'12)*, pages 920–932, 2012.

[59] V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive Runtime Code Scheduling for Heterogeneous Architectures. In *HiPEAC 2009*, pages 19–33, 2009.

[60] M. Joselli, J. Ricardo da Silva, M. Zamith, E. Clua, M. Pelegrino, and E. Mendonca. Techniques for Designing GPGPU Games. In *Proceedings of the 2012 IEEE International Games Innovation Conference (IGIC'12)*, pages 1–5, 2012.

[61] K. Karimi, N. Dickson, and F. Hamze. A Performance Comparison of CUDA and OpenCL. *CoRR*, abs/1005.2581, 2010.

[62] P. Kegel, M. Schellmann, and S. Gorlatch. Comparing Programming Models for Medical Imaging on Multi-core Systems. *Concurrency and Computation: Practice and Experience*, 23(10):1051–1065, July 2011.

[63] Khronos Group. OpenCL—The Open Standard for Parallel Programming of Heterogeneous Systems. http://www.khronos.org/opencl/.

[64] Khronos Group. The OpenCL Specification. https://www.khronos.org/opencl/.

[65] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a Single Compute Device Image in OpenCL for Multiple GPUs. In *PPoPP 2011*, pages 277–288, 2011.

[66] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors - A Hands-on Approach*. Morgan Kaufmann Publishers Inc., 1st edition, 2010.

[67] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer. An Automatic Input-Sensitive Approach for Heterogeneous Task Partitioning. In *ICS 2013*, pages 149–160, 2013.

[68] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi. Evaluating Performance and Portability of OpenCL Programs. In *iWAPT 2010*, 2010.

[69] D. Koufaty and D. T. Marr. Hyperthreading Technology in the Netburst Microarchitecture. *IEEE Micro*, 23(2):56–65, 2003.

[70] J. Lai and A. Seznec. Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13)*, pages 4:1–4:10, 2013.

[71] J. Lee, M. Samadi, Y. Park, and S. A. Mahlke. Transparent CPU-GPU Collaboration for Data-Parallel Kernels on Heterogeneous Systems. In *PACT 2013*, pages 245–255, 2013.

[72] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *PPoPP 2009*, pages 101–110, 2009.

[73] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *ISCA 2010*, pages 451–460, 2010.

[74] P. Li, J. L. Shin, G. Konstadinidis, F. Schumacher, V. Krishnaswamy, H. Cho, S. Dash, R. Masleid, C. Zheng, Y. D. Lin, et al. A 20nm 32-Core 64MB L3 Cache SPARC M7 Processor. In *2015 IEEE International Solid-State Circuits Conference (ISSCC'15)*, pages 1–3, 2015.

[75] Z. Li and Y. Song. Automatic Tiling of Iterative Stencil Loops. *ACM Transactions on Programming Languages and Systems*, 26(6):975–1028, 2004.

[76] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Y. Meng. Merge: A Programming Model for Heterogeneous Multi-core Systems. In *ASPLOS*, pages 287–296, 2008.

[77] E. Lindholm, J. Nickolls, S. F. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008.

[78] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *MICRO 2009*, pages 45–55, 2009.

[79] D. Lustig and M. Martonosi. Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization. In *HPCA 2013*, pages 354–365, 2013.

[80] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang. GreenGPU: A Holistic Approach to Energy Efficiency in GPU-CPU Heterogeneous Architectures. In *ICPP 2012*, pages 48–57, 2012.

[81] S. Madougou, A. L. Varbanescu, C. de Laat, and R. van Nieuwpoort. An Empirical Evaluation of GPGPU Performance Models. In *Euro-Par 2014 Workshops*, pages 165–176, 2014.

[82] Z. Majo and T. R. Gross. A Library for Portable and Composable Data Locality Optimizations for NUMA Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*, pages 227–238, 2015.

[83] K. K. Matam and K. Kothapalli. Accelerating Sparse Matrix Vector Multiplication in Iterative Methods Using GPU. In *ICPP 2011*, pages 612–621, 2011.

[84] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/.

[85] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Frameworks for Multi-core Architectures: A Comprehensive Evaluation Using 2D/3D Image Registration. In *Proceedings of the 24th International Conference on Architecture of Computing Systems (ARCS'11)*, pages 62–73, 2011.

[86] J. Meng and K. Skadron. Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs. In *ICS 2009*, pages 256–265, 2009.

[87] S. Mittal and J. Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys*, 2015.

[88] A. Nere, S. Franey, A. Hashmi, and M. H. Lipasti. Simulating cortical networks on heterogeneous multi-GPU systems. *Journal of Parallel and Distributed Computing*, 73(7):953–971, 2013.

[89] J. Nickolls and W. J. Dally. The GPU Computing Era. *IEEE Micro*, 30(2):56–69, 2010.

[90] Nvidia. CUDA C Programming Guide. http://docs.nvidia.com/cuda/.

[91] Nvidia. NVIDIA OpenCL SDK. https://developer.nvidia.com/opencl.

[92] S. Olivier and J. Prins. Comparison of OpenMP 3.0 and Other Task Parallel Frameworks on Unbalanced Task Graphs. *International Journal of Parallel Programming*, 38(5):341–360, 2010.

[93] OpenACC. The OpenACC Application Programming Interface. http://www.openacc-standard.org, 2013.

[94] OpenMP Architecture Review Board. Open Multi-Processing. http://openmp.org/wp/.

[95] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.0. http://openmp.org/wp/openmp-specifications/, 2013.

[96] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[97] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis. A new era in scientific computing: Domain decomposition methods in hybrid CPU–GPU architectures. *Computer Methods in Applied Mechanics and Engineering*, 200(13):1490–1508, 2011.

[98] I. K. Park, N. Singhal, M. H. Lee, S. Cho, and C. W. Kim. Design and Performance Evaluation of Image Processing Algorithms on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):91–104, 2011.

[99] R. A. Pearce, M. Gokhale, and N. M. Amato. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *SC 2010*, pages 1–11, 2010.

[100] A. Penders. Accelerating Graph Analysis with Heterogeneous Systems. Master's thesis, TU Delft, 2012.

[101] J. Pisharath, Y. Liu, W. Liao, A. Choudhary, G. Memik, and J. Parhi. NU-MineBench 2.0. Technical Report CUCIS-2005-08-01, Northwestern University, 2005.

[102] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Self-Adaptive OmpSs Tasks in Heterogeneous Environments. In *IPDPS 2013*, pages 138–149, 2013.

[103] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal. Compiler and Runtime Support for Enabling Generalized Reduction Computations on Heterogeneous Parallel Configurations. In *ICS 2010*, pages 137–146, 2010.

[104] S. A. Rizzi and B. M. Sullivan. Synthesis of virtual environments for aircraft community noise impact studies. In *the 11th AIAA/CEAS Aeroacoustics Conference*, pages AIAA–2005–2983, 2005.

[105] O. Rosenberg. Optimizing OpenCL on CPUs. http://www.khronos.org/assets/uploads/developers/library/2010_siggraph_bof_opencl/OpenCL-BOF-Intel-SIGGRAPH-Jul10.pdf, 2010.

[106] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, pages 73–82, 2008.

[107] E. Salomons. *Computational atmospheric acoustics*. Kluwer Academic Publishers, 2001.

[108] T. Scogland, B. Rountree, W. Feng, and B. R. de Supinski. Heterogeneous Task Scheduling for Accelerated OpenMP. In *IPDPS 2012*, pages 144–155, 2012.

[109] S. Seo, G. Jo, and J. Lee. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In *IISWC 2011*, pages 137–148, 2011.

[110] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. Performance Gaps between OpenMP and OpenCL for Multi-core CPUs. In *Proceedings of the 41st International Conference on Parallel Processing Workshops (ICPPW'12)*, pages 116–125, 2012.

[111] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. An application-centric evaluation of OpenCL on multi-core CPUs. *Parallel Computing*, 39(12):834–850, 2013.

[112] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. Performance Traps in OpenCL for CPUs. In *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'13)*, pages 38–45, 2013.

[113] J. Shen, J. Fang, A. L. Varbanescu, and H. Sips. OpenCL vs. OpenMP: A Programmability Debate. In *Proceedings of the 16th Workshop on Compilers for Parallel Computing (CPC'12)*, 2012.

[114] J. Shen and A. L. Varbanescu. A Detailed Performance Analysis of the OpenMP Rodinia Benchmark. Technical Report PDS-2011-011, PDS Group, Delft University of Technology, 2012.

[115] J. Shen, A. L. Varbanescu, X. Martorell, and H. Sips. A Study of Application Kernel Structure for Data Parallel Applications. Technical Report PDS-2015-001, PDS Group, Delft University of Technology, 2015.

[116] J. Shen, A. L. Varbanescu, X. Martorell, and H. Sips. Matchmaking Applications and Partitioning Strategies for Efficient Execution on Heterogeneous Platforms. In *Proceedings of the 44th International Conference on Parallel Processing (ICPP'15)*, 2015.

[117] J. Shen, A. L. Varbanescu, and H. Sips. Look before You Leap: Using the Right Hardware Resources to Accelerate Applications. In *Proceedings of the 16th IEEE International Conference on High Performance Computing and Communications (HPCC'14)*, pages 383–391, 2014.

[118] J. Shen, A. L. Varbanescu, H. Sips, M. Arntzen, and D. G. Simons. Glinda: A Framework for Accelerating Imbalanced Applications on Heterogeneous Platforms. In *Proceedings of the ACM International Conference on Computing Frontiers (CF'13)*, pages 14:1–14:10, 2013.

[119] J. Shen, A. L. Varbanescu, P. Zou, Y. Lu, and H. Sips. Efficient Deployment of Data Parallel Applications on Heterogeneous Platforms. *Under review*.

[120] J. Shen, A. L. Varbanescu, P. Zou, Y. Lu, and H. Sips. Improving Performance by Matching Imbalanced Workloads with Heterogeneous Platforms. In *Proceedings of the 28th International Conference on Supercomputing (ICS'14)*, pages 241–250, 2014.

[121] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. M. Badia, and A. Purkayastha. A Framework for Performance Modeling and Prediction. In *SC 2002*, pages 1–17, 2002.

[122] F. Song, S. Tomov, and J. Dongarra. Enabling and Scaling Matrix Computations on Heterogeneous Multi-Core and Multi-GPU Systems. In *ICS 2012*, pages 365–376, 2012.

[123] J. Stone, D. Gohara, and G. Shi. OpenCL: A parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science and Engineering*, 12(3):66, 2010.

[124] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, 2012.

[125] S. Tomov, J. Dongarra, and M. Baboulin. Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. *Parallel Computing*, 36(5-6):232–240, 2010.

[126] A. Vajda. *Programming Many-Core Chips*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[127] B. van Werkhoven, J. Maassen, F. J. Seinstra, and H. E. Bal. Performance models for CPU-GPU data transfers. In *CCGrid 2014*, pages 11–20, 2014.

[128] S. Venkatasubramanian and R. W. Vuduc. Tuned and Wildly Asynchronous Stencil Kernels for Hybrid CPU/GPU Systems. In *ICS 2009*, pages 244–255, 2009.

[129] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure. On the Limits of GPU Acceleration. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism (HotPar'10)*, pages 13–13, 2010.

[130] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson. Comparing Hardware Accelerators in Scientific Applications: A Case Study. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):58–68, 2011.

[131] S. Williams, A. Waterman, and D. A. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[132] M. Wolf. *High-Performance Embedded Computing, Applications in Cyber-Physical Systems and Mobile Computing*. Morgan Kaufmann Publishers Inc., 2nd edition, 2014.

[133] J. Wu and J. JáJá. High Performance FFT Based Poisson Solver on a CPU-GPU Heterogeneous Platform. In *IPDPS13*, pages 115–125, 2013.

[134] L. T. Yang, X. Ma, and F. Mueller. Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution. In *SC 2005*, page 40, 2005.

[135] X. Yang, X. Liao, K. Lu, Q. Hu, J. Song, and J. Su. The TianHe-1A Supercomputer: Its Hardware and Software. *Journal of Computer Science and Technology*, 26(3):344–351, 2011.

[136] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. A Fully Integrated Multi-CPU, GPU and Memory Controller 32nm Processor. In *2011 IEEE International Solid-State Circuits Conference (ISSCC'11)*, pages 264–266, 2011.

Summary

Efficient High Performance Computing on Heterogeneous Platforms

Nowadays, heterogeneous platforms mixing different processing units are popular and interesting options for high performance computing. Such platforms combining different hardware capabilities are likely to provide great performance improvement for data parallel applications. Several studies have already shown that using heterogeneous platforms can improve application performance and hardware utilization. However, efficiently matching platforms with increasing heterogeneity and applications with various patterns and behaviors to achieve high performance is an inherently complex problem. In this thesis, we design and evaluate systematic methods which enable a large variety of data parallel applications to efficiently utilize heterogeneous platforms.

We follow an application-centric approach to study high performance heterogeneous computing, and we decompose our research into three dimensions.

First, in the *application parallelization dimension* (Chapter 2), we study how to efficiently parallelize applications on heterogeneous platforms. To this end, we evaluate the OpenCL programming model as its cross-platform code portability makes it an interesting option for heterogeneous computing. We evaluate OpenCL with multiple applications, and propose generic code transformations to make OpenCL perform well on the different processors of a heterogeneous platform.

Second, in the *application workload dimension* (Chapters 3, 4, 5), we study how to efficiently map application workloads onto heterogeneous platforms. We start from applications with imbalanced workloads in which relatively few data points require more computation than other data points. We propose a workload partitioning framework that matches the heterogeneity of the platform with the imbalance of the workload and applies an auto-tuning method to determine the optimal partitioning (Chapter 3). We further optimize the workload partitioning process by developing a fast prediction method that replaces auto-tuning. This method models workload partitioning and correctly predicts the optimal partitioning (Chapter 4). Finally, we generalize the model-based prediction method for both balanced and imbalanced applications. Although it seems reasonable to accelerate balanced applications on homogeneous platforms like GPUs, we demonstrate that it is still beneficial, performance-wise, to use heterogeneous platforms and to choose

the right hardware configuration (Only-GPU, Only-CPU, or the mix of both) for every application instance, depending on both the available hardware and the input dataset. Therefore, we propose a systematic approach to determine the best hardware configuration and, when needed, the optimal partitioning. We further generalize our approach to heterogeneous platforms with multiple identical or non-identical accelerators (Chapter 5).

Third, in the *application kernel dimension* (Chapter 6), we consider applications with one or multiple kernels and more complex kernel structures, and we study how to efficiently accelerate these applications. Specifically, we propose an application classification and a set of static and dynamic partitioning strategies that provides more than sufficient coverage for each application class; we design an application analyzer to determine the application's class and select the best performing partitioning strategy for that class. In this way, we achieve both high performance and wide applicability for our workload partitioning.

Our research has made the following contributions. (1) We empirically demonstrate the suitability of OpenCL as a unified programming model for heterogeneous computing, and improve its efficiency for programming heterogeneous platforms. (2) We develop a workload partitioning framework to accelerate imbalanced applications on heterogeneous platforms, and empirically demonstrate its effectiveness. (3) We propose a model-based prediction method to correctly and quickly predict the optimal workload partitioning, preserving the performance gain while speeding up the partitioning process. (4) We design a systematic workload partitioning approach which improves performance for both balanced and imbalanced applications, for applications with different datasets and execution scenarios, and for platforms with different hardware mixes. (5) We propose an application classification and a set of partitioning strategies, and we design a matchmaking procedure that matches applications and partitioning strategies to enable efficient execution for most data parallel applications on heterogeneous platforms.

Overall, we conclude (in Chapter 7) that heterogeneous platforms are the right solution, performance-wise, for many classes of data parallel applications. We further propose, based on all our findings, a generic framework that makes efficient heterogeneous computing accessible to many users. Our future research directions on new benchmarking methodologies, integration with distributed systems, extensions to different application classes, and extension of the partitioning model, will further broaden the usability of our work.

Samenvatting

Efficiënte High Performance Computing op Heterogene Platformen

Heterogene platformen die verschillende soorten verwerkingseenheden benutten zijn tegenwoordig populaire en interessante opties voor *high performance computing*. Deze platformen, die verschillende hardware opties combineren, kunnen waarschijnlijk de prestaties van *data parallel applications* aanzienlijk verbeteren. Uit eerdere onderzoeken is al gebleken dat het gebruik van heterogene platformen de prestaties van applicaties en de benutting van hardware kan verbeteren. Het efficiënt combineren van platformen, die steeds heterogener worden, en applicaties, met verschillende patronen en gedragingen, is van nature een complex probleem. In deze thesis ontwerpen en evalueren we systematische methodes die het mogelijk maken vele soorten *data parallel applications* efficiënt gebruik te laten maken van heterogene platformen.

We hanteren een op applicaties gerichte aanpak om heterogeen *high performance computing* te onderzoeken, en delen ons onderzoek op in drie dimensies.

Ten eerste onderzoeken we, in de *application parallelization dimension* (Hoofdstuk 2), hoe applicaties efficiënt geparalleliseerd kunnen worden op heterogene platformen. Dit doen we door het *OpenCL programming model*, wat door zijn *cross-platform code portability* een interessante optie is voor heterogene verwerking, te evalueren. We evalueren OpenCL met meerdere applicaties en presenteren generieke code transformaties die ervoor moeten zorgen dat OpenCL goed presteert op de verschillende processoren van een heterogeen platform.

Ten tweede onderzoeken we, in de *application workload dimension* (Hoofdstuks 3, 4, 5), hoe applicatie werklasten efficiënt toegewezen kunnen worden aan heterogene platformen. Eerst kijken we naar applicaties met ongebalanceerde werklasten waarbij de benodigde berekeningen voor data in deze werklasten relatief weinig onderling verschillen. We presenteren een werklast verdelingsframework dat de heterogeniteit van het platform overeen laat komen met de onbalans van de werklast en dat een *auto-tuning* methode toepast om uiteindelijk de optimale verdeling te bepalen (Hoofdstuk 3). We optimaliseren het werklast verdelingsproces verder door een *fast prediction* methode te ontwikkelen die de *auto-tuning* methode vervangt. Deze methode modelleert het verdelen van werklasten en voorspelt de optimale verdeling (Hoofdstuk 4). Uiteindelijk gener-

aliseren we de op modellen gebaseerde methode voor gebalanceerde en ongebalanceerde applicaties. Het is niet onredelijk om gebalanceerde applicaties te versnellen op homogene platformen, zoals GPUs. Wij laten echter zien dat het nog steeds voordelig is, gelet op de prestaties, om heterogene platformen te gebruiken en daarbij de juiste hardware configuratie (alleen GPU, alleen CPU, of beide) te kiezen, op basis van de beschikbare hardware en de gegeven dataset, voor elke draaiende applicatie. Derhalve presenteren wij een systematische aanpak om de beste hardware configuratie en als dat nodig is, de optimale verdeling te bepalen. We generaliseren onze aanpak verder voor heterogene platformen met meerdere identieke of niet identieke versnellers (Hoofdstuk 5).

Ten derde kijken we, in de *application kernel dimension* (Hoofdstuk 6), naar applicaties met één of meerdere *kernels* en complexere *kernel* structuren en onderzoeken hoe deze efficiënt versneld kunnen worden. Specifiek presenteren we een classificatie van applicaties en een set statische en dynamische verdelingsstrategieën die deze applicatie klassen meer dan afdoende dekt. Ook ontwerpen we een applicatie analysator om de applicatie klasse te bepalen en de verdelingsstrategie te kiezen met de beste prestaties voor die klasse. Op deze manier bereiken we zowel hoge prestaties als brede toepasbaarheid van onze werklast verdeler.

Ons onderzoek heeft de volgende bijdragen opgeleverd. (1) We hebben empirisch de geschiktheid van OpenCL als algemeen geldig *programming model* voor heterogene verwerking gedemonstreerd, en verbeteren de efficiëntie voor programmeren op heterogene platformen. (2) We ontwikkelen een werklast verdelingsframework om ongebalanceerde applicaties op heterogene platformen te versnellen, en demonstreren empirisch de effectiviteit. (3) We presenteren een op modellen gebaseerde voorspellingsmethode om correct en snel de optimale werklast verdeling te voorspellen, met deze methode leveren we niks in op de prestatie winst terwijl het verdelingsproces wordt versneld. (4) We ontwerpen een systematische werklast verdelingsaanpak die de prestaties voor gebalanceerde en ongebalanceerde applicaties, voor applicaties met verschillende datasets en uitvoeringsscenario's en voor platformen met verschillende hardware samenstellingen, verbetert. (5) We presenteren een applicatie classificatie en een set verdelingsstrategieën, en ontwerpen een procedure die applicaties overeen laat komen met verdelingsstrategieën om efficiënte uitvoering voor de meeste *data parallel applications* op heterogene platformen mogelijk te maken.

Alles in acht nemend concluderen we (in Hoofdstuk 7) dat heterogene platformen de juiste oplossing zijn, gelet op de prestaties, voor een groot aantal klassen van *data parallel applications*. Verder presenteren we, gebaseerd op onze resultaten, een generiek framework dat heterogene verwerking toegankelijk maakt voor veel gebruikers. Ons toekomstige onderzoek gericht op *benchmarking* methodologieën, integratie met gedistribueerde systemen, uitbreidingen naar andere applicatie klassen, en uitbreidingen van het verdelingsmodel, zullen de bruikbaarheid van ons werk verder verbreden.

Biography

Jie Shen was born in Changsha, China, on May 18, 1987. In 2005, Jie started her study at National University of Defense Technology (NUDT), Changsha, in the School of Computer. In 2009, she received her B.Eng. degree in Network Engineering, graduating as Excellent Graduate Student (top 2%). Because of her performance, Jie was recommended to start her master program in Computer Science and Technology without sitting the entrance examination. In the next 2 years, she finished her master study at NUDT and was recommended to start her PhD program ahead of master graduation time. In 2011, Jie was awarded a 4-year scholarship from the China Scholarship Council (CSC) to pursue her PhD overseas. In September 2011, she came to the Netherlands, and became a PhD student in the Parallel and Distributed Systems (PDS) group of Delft University of Technology (TU Delft), under the supervision of Prof. Henk Sips and Dr. Ana Lucia Varbanescu. In 2014, she received a 3-month HiPEAC collaboration grant and was a visiting researcher at the Barcelona Supercomputing Center (Spain), working under the supervision of Dr. Xavier Matorell. Between 2013 and 2015, she also worked as a teaching assistant for the Parallel Algorithms and Parallel Computers course at TU Delft.

Jie's main research interests are in parallel computing on multi-core and many-core processors (CPUs, GPUs, and heterogeneous platforms), with a focus on performance, portability, and programmability of many-core heterogeneous systems.

Publications

International (refereed) journals

1. **Jie Shen**, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. An application-centric evaluation of OpenCL on multi-core CPUs. *Parallel Computing*, 39(12):834–850, 2013.

2. **Jie Shen**, Ana Lucia Varbanescu, Yutong Lu, Peng Zou, and Henk Sips. Efficient Deployment of Data Parallel Applications on Heterogeneous Platforms. *Under review*.

International (refereed) conferences

3. **Jie Shen**, Ana Lucia Varbanescu, Xavier Martorell, and Henk Sips. Matchmaking Applications and Partitioning Strategies for Efficient Execution on Heterogeneous Platforms. In *Proceedings of the 44th International Conference on Parallel Processing (ICPP'15)*, 2015.

4. **Jie Shen**, Ana Lucia Varbanescu, and Henk Sips. Improving Application Performance by Efficiently Utilizing Heterogeneous Many-core Platforms. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid'15)*, 2015.

5. **Jie Shen**, Ana Lucia Varbanescu, and Henk Sips. Look before You Leap: Using the Right Hardware Resources to Accelerate Applications. In *Proceedings of the 16th IEEE International Conference on High Performance Computing and Communications (HPCC'14)*, 2014.

6. **Jie Shen**, Ana Lucia Varbanescu, Peng Zou, Yutong Lu, and Henk Sips. Improving Performance by Matching Imbalanced Workloads with Heterogeneous Platforms. In *Proceedings of the 28th International Conference on Supercomputing (ICS'14)*, 2014.

7. **Jie Shen**, Ana Lucia Varbanescu, Henk Sips, Michael Arntzen, and Dick G. Simons. Glinda: A Framework for Accelerating Imbalanced Applications on Heterogeneous Platforms. In *Proceedings of the ACM International Conference on Computing Frontiers (CF'13)*, 2013.

8. **Jie Shen**, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. Performance Traps in OpenCL for CPUs. In *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'13)*, 2013.

9. **Jie Shen**, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. Performance Gaps between OpenMP and OpenCL for Multi-core CPUs. In *Proceedings of the 41st International Conference on Parallel Processing Workshops (ICPPW'12)*, 2012.

Co-Authored Publications

10. Jianbin Fang, Ana Lucia Varbanescu, **Jie Shen**, and Henk Sips. ELMO: A User-Friendly API to Enable Local Memory in OpenCL Kernels. In *Proceedings of the 21st International Conference on Parallel, Distributed, and Network-Based Processing (PDP'13)*, 2013.

11. Michael Arntzen, **Jie Shen**, Ana Lucia Varbanescu, Henk Sips, and Dick G. Simons. Acoustic Ray Tracing Parallelization. In *INTER-NOISE and NOISE-CON Congress and Conference Proceedings (Noise-Con'13)*, 2013.

12. Jianbin Fang, Ana Lucia Varbanescu, **Jie Shen**, Henk Sips, Gorkem Saygili, and Laurens van der Maaten. Accelerating Cost Aggregation for Real-Time Stereo Matching. In *Proceedings of the 18th IEEE International Conference on Parallel and Distributed Systems (ICPADS'12)*, 2012.